University of Piraeus

School of Information and Communication Technologies

Department of Digital Systems

Postgraduate Program of Studies

MSc Digital Systems Security

MSc Dissertation

Creation of an Android Security Training Lab

Supervisor Professor: Xenakis Christos

Name-Surname	E-mail	Student ID.
Grigoris Papoutsis	gpapoutsis@ssl-unipi.gr	MTE 1927

Piraeus

28/02/2022

Copyright © Papoutsis Grigorios, 2022 – All rights reserved

It is prohibited to copy, store and distribute this work, in whole or in part, for commercial purposes. Reproduction, storage and distribution for non-profit, educational or research purposes is permitted provided the source of origin is referenced and the present message maintained.

Questions about the use of work for profit should be addressed to the author (bertolis24@hotmail.com).

This document reflects the results of a study that has been prepared on behalf of the Postgraduate Program "Digital Systems Security" at University of Piraeus. The information and conclusions contained in this thesis express the author's personal opinion and arguments, and therefore should not be interpreted that they represent the official concepts of University of Piraeus.

Abstract

Mobile attack incidents have increased in recent years both on enterprise and personal level. One way to fight this is keeping our security teams up to date with the latest trends, and have security awareness as individuals. In order to achieve this goal, this thesis will provide a comprehensive methodology on how to perform Android application security assessment. It will teach the reader what are the fundamental things that someone needs to know before starting the Android app assessment, explain in depth some of the most common techniques that are used, and give a full hands-on experience on the latest mobile security trends, through an immersive gamified Android Application Security Lab. The topics that we will study on this thesis include information gathering and local storage enumeration, reverse engineering, static and dynamic analysis methods, traffic analysis methods and Android forensics. Usage of various tools and setting up operating systems and virtual environments are going to be described as well. Finally, we will show how to configure and deploy an open source CTF web-based platform using the latest technologies like docker, in order for anyone to be able to create their own security lab. To write this document, it was necessary to have a solid understanding of the needs for training, the Android OS structure, the structure of the Android applications and how they are compiled and archived in different file types, the various programming languages that are used for the creation of the challenges and the other platforms that involved, the various and different security assessment methodologies, the technologies, services and virtual environment that were set up, as well as the vulnerabilities and bad practices that were incorporated in the challenges, and examined later in depth when assessing them. As the problem has grown, the security training companies have already paid attention on how to provide a good training content. Similar security training with the one we are going to see in this project has already been provided online. However, while most of the existing online projects are providing mostly theoretical content, this project gives a fully hands-on gamified experience, in a comprehensive and methodical way. To create this lab, the following technologies/services and opensource projects were used: Android Studio, AVD Emulator, CTFd platform and Docker. The programing languages that needed were Java and C++. Finally, the operating systems that used on this project were Parrot Linux and Android OS. In order to read this thesis and play the Lab, one should know the fundamentals of cyber security, have a good understanding of Linux systems and be able to handle command line tools, be familiar with an objective programming language like Java or C++, and be passionate about mobile application security assessment. On completion of this Lab,

one should be able to understand the need for cyber security training, the structure of an android phone and apps, how to assess android applications, how to use tools to automate the assessment, how to do Android forensics, and how to create detailed writeups when completing a CTF security challenge.

Table of Contents

Abstract *4*

Introduction *13*

Android Operating System & Applications *15*

Android OS **15**

Android Apps **15**

About Training Labs & CTFs *17*

Online Training Platforms **17**

CTF Events **18**

Jeopardy 19

Attack Defense 19

King of The Hill 20

Linear 20

Mixed 20

CTF Categories 20

Writeups 21

Assessment Techniques *23*

Enumerating the Local Storage **23**

Important directories 23

Installation directory 24

Extracting Readable Files from the APK 26

Performing Static and Dynamic Analysis **28**

Static Analysis 28

Dynamic Analysis 30

Extracting an APK File 31

Capturing HTTP Requests **34**

Rooting the Device and Acquiring a Disk Image **35**

Rooting the device 35

Acquiring the disk image 36

Setting up the Environment *39*

Android Emulator **39**

Android Debug Bridge **44**

Operating Systems and Tools **47**

Installation 47

Preparation 48

Setting up the CTF Platform ***49***

Installation **50**

Challenges Walkthroughs ***58***

Enumeration **59**

Enumeration01 59

Enumeration02 66

Enumeration03 73

Reverse **79**

Reverse01 79

Reverse02 86

Reverse03 97

Reverse04 110

Traffic Analysis **124**

TrafficAnalysis01 124

Forensics **132**

Forensics01 132

Forensics02 139

Conclusion ***143***

References ***144***

Table of Figures

Figure 1 Market Share 13
Figure 2 SQLite3 25
Figure 3 Shared Prefs 26
Figure 4 AndroidManifest.xml 27
Figure 5 JADX 29
Figure 6 APKTool smali code 29
Figure 7 Firda Native Function Hooking 30
Figure 8 APK Extractor Third Party Tool 32
Figure 9 ADB grep app name 33
Figure 10 ADB pm path 33
Figure 11 ADB pull apk 34
Figure 12 Burp Suit get request interception 35
Figure 13 Root Checker app 36
Figure 14 List Mounted Drivers 37
Figure 15 Disk Image Acquisition 38
Figure 16 Autopsy Home Screen 38
Figure 17 Android Studio Downloading Components 41
Figure 18 Android Studio New Project 42
Figure 19 Android Studio AVD Manager 42
Figure 20 Android Studio Start Device 43
Figure 21 AVD Emulator 43
Figure 22 ADB Installation 45
Figure 23 Enable USB Debagging 46
Figure 24 ADB List Emulators 46
Figure 25 Docker Engin Installation 50
Figure 26 Docker Compose Installation 51
Figure 27 Docker Compose Version 51
Figure 28 Cloning CTFd Platform 52
Figure 29 Starting CTFd Services 52
Figure 30 Docker Listing Services 53
Figure 31 CTFd Suet Up Page 54
Figure 32 Android Security Training Lab Home Page 54

Figure 33 CTFd Admin Panel.....55
Figure 34 CTFd Challenges.....55
Figure 35 CTFd Create New Challenge.....56
Figure 36 CTFd Challenges Properties.....56
Figure 37 CTFd Download Challenges.....57
Figure 38 Enumeration01 ADB List Devices.....60
Figure 39 Enumeration01 Unzip APK.....60
Figure 40 Enumeration01 App.....61
Figure 41 Enumeration01 Download APKTool.....62
Figure 42 Enumeration01 Decompiling APK.....62
Figure 43 Enumeration01 APKTool Error.....63
Figure 44 Enumeration01 List Directory.....63
Figure 45 Enumeration01 Flag.....64
Figure 46 Enumeration02 ADB List Devices.....67
Figure 47 Enumeration02 Unzip APK.....67
Figure 48 Enumeration02 App.....68
Figure 49 Enumeration02 ADB root.....69
Figure 50 Enumeration02 grep App Name.....69
Figure 51 Enumeration02 Listing App Installation Directory.....70
Figure 52 Enumeration02 Database.....70
Figure 53 Enumeration02 SQLite3.....71
Figure 54 Enumeration02 Tables.....71
Figure 55 Enumeration02 Flag.....71
Figure 56 Enumeration03 List Devices.....74
Figure 57 Enumeration03 Unzip APK.....74
Figure 58 Enumeration03 App.....75
Figure 59 Enumeration03 ADB root.....76
Figure 60 Enumeration03 grep App.....76
Figure 61 Enumeration03 shared_prefs Directory.....77
Figure 62 Enumeration03 SharedPreferences.xml.....77
Figure 63 Enumeration03 Flag.....78
Figure 64 Reverse01 ADB List Devices.....80
Figure 65 Reverse01 Unzip APK.....80
Figure 66 Reverse01 App.....81

Figure 67 Decompress APK File 82
Figure 68 Reverse01 DEX2JAR 83
Figure 69 Reverse01 JADX 83
Figure 70 Reverse01 MainActivity 84
Figure 71 Reverse01 LoginActivity 84
Figure 72 Reverse01 Flag 85
Figure 73 Reverse01 Mitigation 85
Figure 74 Reverse02 ABD List Devices 86
Figure 75 Reverse02 Unzip APK 87
Figure 76 Reverse02 App 87
Figure 77 Reverse02 MainActivity 88
Figure 78 Reverse02 Download APKTool 89
Figure 79 Reverse02 APKTool Decompile 90
Figure 80 Reverse02 Listing Decompressed APK 90
Figure 81 Reverse02 Smali Files 91
Figure 82 Reverse02 Smali Code 91
Figure 83 Reverse02 APKTool Recompile 92
Figure 84 Reverse02 New APK 93
Figure 85 Reverse02 Signing Certificate 93
Figure 86 Reverse02 Signing The New APK 94
Figure 87 Reverse02 Uninstalling Old App 94
Figure 88 Reverse02 Installing New App 95
Figure 89 Reverse02 Flag 95
Figure 90 Reverse03 ADB Listing Devices 97
Figure 91 Reverse03 Unzip APK 98
Figure 92 Reverse03 App 98
Figure 93 Reverse03 MainActivity 99
Figure 94 Reverse03 Google Search JNI 100
Figure 95 Reverse03 APKTool Download 101
Figure 96 Reverse03 Decompile APK 101
Figure 97 Reverse03 Share Library 102
Figure 98 Reverse03 Shared Object Content 102
Figure 99 Reverse03 Ghidra Installation 103
Figure 100 Reverse03 Ghidra New Project 103

Figure 101 Reverse03 Ghidra Project Properties 104
Figure 102 Reverse03 Start Project 104
Figure 103 Reverse03 Load File 105
Figure 104 Reverse03 Ghidara File Properties 105
Figure 105 Reverse03 Ghidra Analyze 106
Figure 106 Reverse03 Ghidra Analazy Options 106
Figure 107 Reverse03 Ghidra Summary 107
Figure 108 Reverse03 CodeBrowser 107
Figure 109 Reverse03 Flag 108
Figure 110 Reverse03 Login Screen 109
Figure 111 Reverse04 ADB List Devices 111
Figure 112 Reverse04 Unzip APK 111
Figure 113 Reverse04 App 112
Figure 114 Reverse04 DEX2JAR 112
Figure 115 Reverse04 JADX 113
Figure 116 Reverse04 Download APKTool 114
Figure 117 Reverse04 APKTool Decompile 114
Figure 118 Reverse04 Shared Object Content 115
Figure 119 Reverse04 Download Frida Server 116
Figure 120 Reverse04 Decompress Frida Server 116
Figure 121 Reverse04 ADB push frida server 117
Figure 122 Reverse04 Install Frida Tools 117
Figure 123 Reverse04 JNI Frida Hook 118
Figure 124 Reverse04 Git Clone Hooking Script 118
Figure 125 Reverse04 Hooking Script Variables 119
Figure 126 Reverse04 Script Library and Function Names 119
Figure 127 Reverse04 Script Find Functions 119
Figure 128 Reverse04 Install Frida Compile 120
Figure 129 Reverse04 Get Package Name 120
Figure 130 Reverse04 Start Frida Server 121
Figure 131 Reverse04 Frida Hook Native Functions 121
Figure 132 Reverse04 Hooked Functions 122
Figure 133 Reverse04 Script Function Name 122
Figure 134 Reverse04 Flag 123

Figure 135 TrafficAnalysis ADB List Devices 125
Figure 136 TrafficAnalysis Unzip APK 125
Figure 137 TrafficAnalysis App 126
Figure 138 TrafficAnalysis Get Local IP 127
Figure 139 TrafficAnalysis Burp Proxy Tab 127
Figure 140 TrafficAnalysis Burp Binding IP 128
Figure 141 TrafficAnalysis Burp Intercept On 128
Figure 142 TrafficAnalysis Emulator Settings 129
Figure 143 TrafficAnalysis Emulator Proxy Tab 130
Figure 144 TrafficAnalysis Flag 130
Figure 145 Forensics01 Start Autopsy 133
Figure 146 Forensics01 Case Name and Number 133
Figure 147 Forensics01 Host and Source Type 134
Figure 148 Forensics01 Import File 134
Figure 149 Forensics01 Configute Ingest 135
Figure 150 Forensics01 Autopsy Home Screen 135
Figure 151 Forensics01 Communications Window 136
Figure 152 Forensics01 App Database 136
Figure 153 Forensics01 Extract Database 137
Figure 154 Forensics01 Open Database 137
Figure 155 Forensics01 Get Tables 138
Figure 156 Forensics01 Flag 138
Figure 157 Forensics02 Start Autopsy 140
Figure 158 Forensics02 Home Screen 140
Figure 159 Forensics02 Web History 141
Figure 160 Forensics02 Extract Deleted File 141
Figure 161 Forensics02 Decompress Extracted File 142
Figure 162 Forensics02 Flag 142

Introduction

Mobile applications have become a part of our daily life. We use them on a personal level like paying when we buy something to eat or do shopping, but they also used from companies in order to make processes and communication better between their employees. That means that mobile applications hold and share personal and sensitive information. Thus, applications security is a matter that must be taken under serious consideration.

Searching online for Analytics on Android OS, we can find Statcounter's latest publication about Mobile Operating System Market Share Worldwide [1].

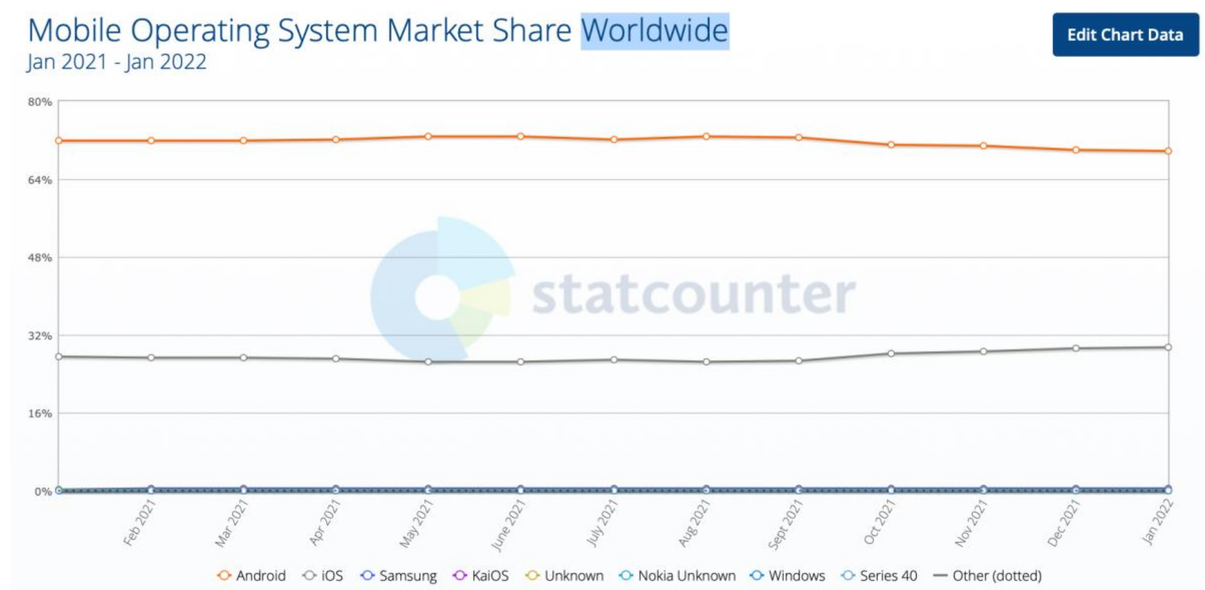


Figure 1 Market Share

As the chart shows, Android OS holds the biggest share of the market, while together with iOS they hold the 99 percent of the market Worldwide.

The move to mass remote working during the COVID-19 pandemic have made mobile devices a fundamental business tool as remote employees are increasingly using their smartphones to access corporate assets and perform critical work tasks. And this trend is about to increase.

In 2020, 97% of organizations faced mobile threats that used various attack vectors. 46% of organizations had at least one employee download a malicious mobile application.

According to Check Point's Mobile Security Report 2021, these are some statistics regarding mobile attacks [2].

- **All enterprises at risk from mobile attacks**: Almost every organization experienced at least one mobile malware attack in 2020. 93% of these attacks originated in a device network attempting to trick users into installing a malicious payload via infected websites or URLs, or to steal users' credentials.
- **Nearly half of organizations impacted by malicious mobile apps**: 46% of organizations had at least one employee download a malicious mobile application that threatened their organization's networks and data in 2020.
- **Four in ten mobiles globally are vulnerable**: At least 40% of the world's mobile devices are inherently vulnerable to cyberattacks due to flaws in their chipsets, and need urgent patching.
- **Mobile malware on the rise**: In 2020, Check Point found a 15% increase in banking Trojan activity, where users' mobile banking credentials are at risk of being stolen. Threat actors have been spreading mobile malware, including Mobile Remote Access Trojans (MRATs), banking Trojans, and premium dialers, often hiding the malware in apps that claim to offer COVID-19 related information.
- **APT groups target mobile devices**: Individuals' mobiles are a very attractive target for various APT groups.

In order to mitigate problems like these, more emphasis should be given on developing application using secure practices. Another factor that will help mitigating this problem, is the application security assessment, which will also be the topic of this thesis.

Before we start analysing the technical part of the Android application security assessment, we first need to understand some basic things about android.

Android Operating System & Applications

Android OS

Android is a mobile operating system based on a modified version of the Linux kernel and other open source software, designed primarily for touchscreen mobile devices such as smartphones and tablets [3].

Android is using the Dalvik Virtual Machine to run applications. Dalvik is a discontinued process virtual machine (VM) in Android operating system that executes applications written for Android.

Programs for Android are commonly written in Java and compiled to bytecode for the Java virtual machine, which is then translated to Dalvik bytecode and stored in .dex (Dalvik EXecutable) and .odex (Optimized Dalvik EXecutable) files. The compact Dalvik Executable format is designed for systems that are constrained in terms of memory and processor speed.

The successor of Dalvik is Android Runtime (ART), which uses the same bytecode and .dex files. The new runtime environment was included for the first time in Android 4.4 "KitKat" as a technology preview, and replaced Dalvik entirely in later versions [4].

Android Apps

Android apps can be written using Kotlin, Java, and C++ languages. The Android SDK tools compile your code along with any data and resource files into an APK or an Android App Bundle.

An Android package, which is an archive file with an .apk suffix, contains the contents of an Android app that are required at runtime and it is the file that Android-powered devices use to install the app.

Each Android app lives in its own security sandbox, protected by the following Android security features:

- The Android operating system is a multi-user Linux system in which each app is a different user.
- By default, the system assigns each app a unique Linux user ID (the ID is used only by the system and is unknown to the app). The system sets permissions for all the files in an app so that only the user ID assigned to that app can access them.
- Each process has its own virtual machine (VM), so an app's code runs in isolation from other apps.
- By default, every app runs in its own Linux process. The Android system starts the process when any of the app's components need to be executed, and then shuts down the process when it's no longer needed or when the system must recover memory for other apps.

It is also interesting to know that the Android system implements the principle of least privilege. That is, each app, by default, has access only to the components that it requires to do its work and no more [5].

Android apps are available on Google Play Store and also on other online websites. Google Play Store application allows the users only to install the app on the phone, while the online websites provide the APK file, so the users can install it by themselves.

An APK (Android Package) file is an app created for Android. Some apps come pre-installed on Android devices, while other apps can be downloaded from Google Play. As we said earlier, apps downloaded from Google Play are automatically installed on your device, while those downloaded from other sources must be installed manually. Typically, users never see APK files because Android handles app installation in the background via Google Play or another app distribution platform [6].

APKs are archived files that among other files, contain the Java classes of the program in a singe dex file. As for the dex files, they are executable files saved in a format that contains compiled code written for Android. It is technically referred to as a "Dalvik Executable," and can be interpreted by the Dalvik virtual machine [7].

About Training Labs & CTFs

Online training is something that we see a lot these years. Knowledge is transferred through the internet, from anywhere in the globe to targeted audiences who choose to learn a particular subject. Some of the platforms provide free content, while on others you have to pay. Some of them also provide certificates of completion. Notes in PDFs, Word documents, video tutorials, assessments or hands-on exercises are given as a package with the training module.

In this section, we are going to see different types of online training that is provided on the Mobile Security field.

Online Training Platforms

We have seen offensive cyber security training offered in many online platforms, whether as paid courses, or as gamified challenges. However not all of them are providing content for mobile and specifically Android training.

Some of the platforms one could find online that provide Android security training content are the following:

- Hack The Box
    - Hack The Box is a massive, online cybersecurity training platform, allowing individuals, companies, universities and all kinds of organizations around the world to level up their hacking skills. HTB recently included Intro to Android Exploitation Track.
- Try Hack Me
    - TryHackMe is a free online platform for learning cyber security, using hands-on exercises and labs, all through a browser. Android Mobile Application Penetration Testing section is included.
- Hacker101**:**
    - Hacker101 is a free class for web security. However, they also have included Mobile Hacking Content.

- DIVA Android**:**
    - DIVA (Damn insecure and vulnerable App) is an App intentionally designed to be insecure. The aim of the App is to teach developers/QA/security professionals, flaws that are generally present in the Apps due poor or insecure coding practices.
- Udemy:
    - Udemy is an online learning and teaching marketplace with over 183000 courses and 40 million students. Mobile Application Security and Penetration Testing is a course that teaches security issues in mobile applications & devices, and penetration testing. Udemy also provides a certificate of completion.
- Cybrary:
    - Cybrary is a Cyberecurity training online platform that among other, it provides the Mobile App Security course that teaches app security testing. Cybrary also provides a certificate of completion.

Some of these are platforms that apart from theoretical content also offer hands on experience.

CTF Events & Types

Another, and kind of different way to provide training in the cyber security field, is through Capture The Flag (CTF) events.

CTFs in computer security is an exercise in which "flags" are secretly hidden in purposefully-vulnerable programs or websites. Competitors steal flags either from other competitors (attack/defense-style CTFs) or from the organizers (jeopardy-style challenges). Competitions exist both online and in-person, and can be advanced or entry-level [8].

CTFs are of different types and consists of different categories. Let’s see some of the basic types of a CTF event

Jeopardy

In a Jeopardy contest, the participants are individuals or teams who are called upon to solve a series of challenges that are usually in different categories to cover as much as possible the research field of Information Security in a limited or not timeframe.

In addition, Jeopardy contests give participants the opportunity to focus on challenges of their choice where they are more familiar with that specific field or think that it's more entertaining.

Each player or team, by the time they solve a challenge, they submit the flag on a scoring board provided by the organizers and acquire the corresponding points for resolve it. Winner is the one who accrues most points.

Contests of this type are the most common as they have a low degree of complexity, require less preparation and configuration than other types. They also require simpler software and hardware infrastructure, can be monitored and rated more easily, and allow a large number of groups or individuals to participate.

Attack Defense

In an Attack – Defense scenario, players are divided into groups. Each team is given by the organizers one or more servers with weaknesses and hidden flags. Its role is to effectively defend its systems and to identify and patch on time the weaknesses that exist in the system in order to repel the attacks of the opposing groups aimed at obtaining the flags. The team's primary responsibility is to take on a defensive role, also known as the blue teaming.

The team also has an aggressive role, known as red teaming, that is to attack the systems of the rival teams in order to violate their servers and to intercept the flags.

This type of contests are more demanding for the organizers. They have greater complexity than Jeopardy, higher infrastructure requirements, more difficult configuration and more complex assessment so that is why they are usually organized locally rather than online.

King of The Hill

In the King of the Hill category there are multiple vulnerable Servers ready to get exploited that do not belong to any group.

The teams are called upon to break vulnerable servers and if they do, the first team is rewarded with the original conquering points to acquire the server. They are then asked to defend this site from rival teams by patching the vulnerabilities. The team that manages to break into the server and then maintain access to it is rewarded with the most points.

Linear

Linear contests, a not-so-common type of contest, are based on challenges that need to be solved in a linear order. Typically, the challenges are narrative and present a story with multiple challenges that need to be solved mandatory in a specific order.

Such competitions are usually organized by companies aimed at finding competent employees who wish to prove their abilities

Mixed

The Mixed category may include features from Jeopardy and Attack – Defense competitions where participants are asked to solve a set of predefined challenges from the organizers but at the same time to have an aggressive and defensive role on opposing teams [9].

CTF Categories

As we said earlier, CTF events also have categories. Some examples of categories are:

- Cryptography
- Stenography
- Binary Exploitation
- Web Exploitation
- Forensics

- Reverse Engineering
- Programming
- Packet Analysis
- Miscellaneous
- Mobile
- Hardware
- Blockchain

In addition, each category has a difficulty rating so that the contest is eligible for participants with different backgrounds.

A CTF-like laboratory is the project that we are describing in the current paper. This CTF-like laboratory will consist only of Mobile challenges divided in subcategories, while it will follow the Jeopardy contest standards.

Some Examples of such CTFs are the qualifying rounds of the known DEFCON CTF as well as the NYU Polytechnic Institutes Cyber Security Awareness (CSAW). Another big CTF event created for universities, is Hack The Box's UNI CTF.

Another thing that is worth to mention, is CTFTime. It is a website that contains all the details of an upcoming online CTF event, while it provides team ranking according to their overall CTF score.

Writeups

When a team or an individual has successfully completed a challenge, they are usually tasked to create the writeup as well. Writeups are something like the walkthrough of the challenge, but at the same time they have to be precise and detailed as an assessment report. Writeups are essentially explaining how the flag was actually captured.

Writeups also can be used from beginners, so they can easily start their journey in the offensive cyber security field, since many of the enthusiasts are struggling to find a starting point.

Both individuals and companies can benefit from participating in such competitions, as individuals can learn new things in an entertaining and fun way, and enterprise teams can keep up with the latest security trends and continue to improve their skills.

Assessment Techniques

There are many techniques one could use in order to test Android applications. In this documentation we will examine applications using both static and dynamic analysis methods.

Reverse engineering, traffic analysis, or even basic enumeration of an apk file and its installation directory, could lead to potential data exposure that in turn can be crucial for the intended function of the application itself.

The assessment methodology that is followed in order to solve the challenges of the Android Security Training Lab, is structured like this.

- Information Gathering
- Reversing
- Static Analysis
- Dynamic Analysis
- Report

In this section we are going to discuss, some of the most common methods that are used in Android application security assessments.

Enumerating the Local Storage

Enumeration is a process that one could start with when assessing Android applications. It is a process that can reveal clues that are necessary in order to further continue assessing the app. This stage needs only a few tools to be used, but it also needs to have a deep knowledge and good understanding of the Android and applications structure.

Important directories

On an Android device, there are several directories that are important to us while conducting an assessment. The directories that are listed below are some of the most important ones [10].

- /**data/data**: This directory contains all the applications that are installed by the user.
- **/data/user/0**: This directory contains data that only the app can access.
- **/data/app**: This directory stores the APKs of the applications that are installed by the user.
- **/system/app**: This directory contains the pre-installed applications of the device.
- **/system/bin**: This directory contains binary files.
- **/data/local/tmp**: This is a world writable directory.
- **/data/system**: This directory contains system configuration files.
- **/etc/apns-conf.xml**: This file contains the default Access Point Name (APN) configurations. APN is used in order for the device to connect with our current carrier's network.
- **/data/misc/wifi**: This directory contains WiFi configuration files.
- **/data/misc/user/0/cacerts-added**: User certificate store. This directory stores certificates added by the user.
- **/etc/security/cacerts/**: System certificate store. Permission to non-root users is not permitted.
- **/sdcard**: This directory contains a symbolic link of the directories DCIM, Downloads, Music, Pictures, etc.

Installation directory

In order to enumerate the file structure of an installed application, we need to have access to a real or emulated device. This could be achieved by using the Android Debug Bridge (ADB). ADB is a command-line tool that lets you communicate directly with the device. Using ADB, we can install and debug applications, while the Unix shell that is provided can be used to run commands on the device. A more in-depth description about ADB will be provided in the next chapter of this documentation.

Installed applications in Android are stored in the **/data/data/** directory. There, are stored all the files that are necessary for the app to work. This directory is private and cannot be accessed by other apps or by non-root users. Android Debug Bridge (ADB) allows as to read the content of this directory, since it provides access through a Unix shell on the device as root.

Having access to this directory we are allowed to search the app for common mistakes and bad practices that programmers do. Such mistakes include leaving unencrypted information inside databases or other files and structures that are used from the app to save data.

Assuming that the installed application with the package name **com.example.myapp**, is using an unencrypted database named **myappdb.db3**, to keep its users login credentials. This database will be stored under **/data/data/com.example.myapp/databases/myappdb.db3**. Having access via adb, we can fetch the content of the database's columns, using the pre-installed **mysqlite3** client.

```
sqlite3 /data/data/com.example.myapp/databases/myappdb.db3
sqlite> .tables
sqlite> select * from Passwords;
```

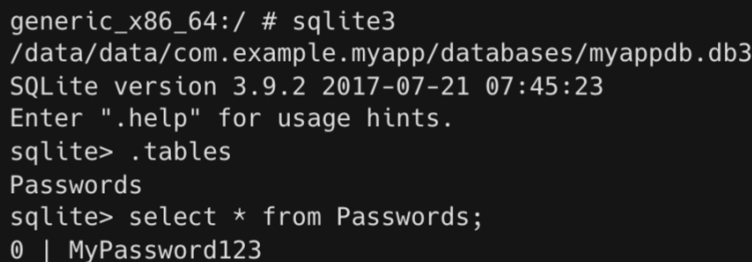


Figure 2 SQLite3

Another example of searching for exposed sensitive data in the application's installation directory, is when the app is using unencrypted key-value pairs in SharedPreferences xml files. These files are used by the app to save small collections of key-value pairs in xml files.

Assuming that we have the same app install in an Android device as in the previous example, only this time the app stores its data in the **preferences.xml** file. Then the SharedPreferences file could be read by someone by accessing the directory

/data/data/com.example.myapp/shared_prefs/ and listing the content of the **preferences.xml** file.

```
cat /data/data/com.example.myapp/shared_prefs/preferences.xml
```

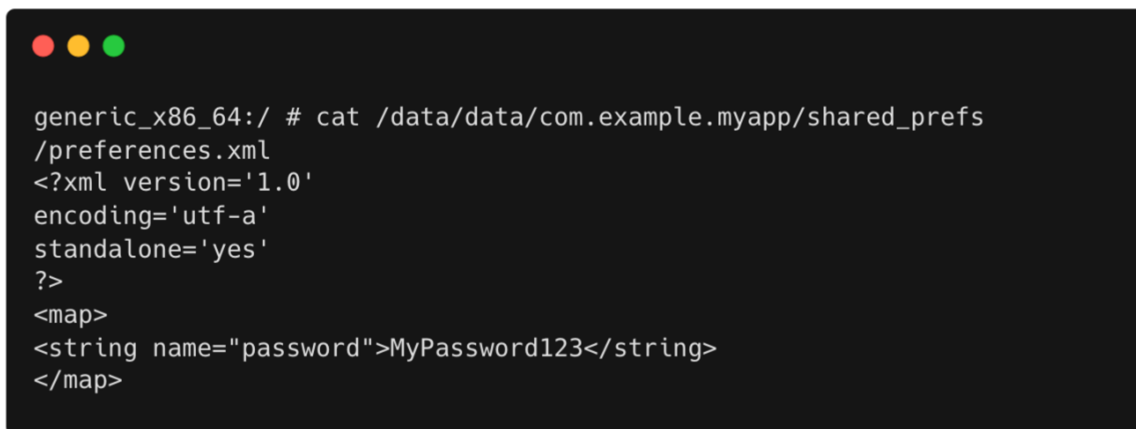


Figure 3 Shared Prefs

Extracting Readable Files from the APK

As we will explain later on the next chapters, an APK (Android Packages) file is an app created for Android. They are saved in a compressed .ZIP format and can be opened by any Zip decompression tool.

Android packages contain all the necessary files for a single Android program. Below is a list of the most prominent files and folders [11]:

- **META-INF/**: Contains the manifest file, signature, and a list of resources in archive
- **lib/**: Native libraries that run on specific device architectures (armeabi-v7a, x86, etc.)
- **res/**: Resources, such as images, that were not compiled into resources.arsc
- **assets/**: Raw resource files that developers bundle with the app
- **AndroidManifest.xml**: Describes the name, version, and contents of the APK file
- **classes.dex**: The compiled Java classes to be run on the device (.DEX file)
- **resources.arsc**: The compiled resources, such as strings, used by the app (.ARSC file)

Some of these files will be examined while performing other assessing methods on next chapters, like reverse engineering. **AndroidManifest.xml** is a file that is definitely worth to check.

Every app project must have an AndroidManifest.xml file (with precisely that name) at the root of the project source set. The manifest file describes essential information about the app to the Android build tools, the Android operating system, and Google Play [12].

- The components of the app, which include all activities, services, broadcast receivers, and content providers
- The permissions that the app needs in order to access protected parts of the system or other apps
- The hardware and software features the app requires, which affects which devices can install the app from Google Play

Bad security practices are often implemented in **AndroidManifest.xml** and thus, this is one of the first things pentesters are usually check when assessing an Android app.

Unzipping an apk file to read the **AndroidManifest.xml** file is not enough, as the file is encoded. Tools like Apktool can decode this file and make it human readable. This is how a decoded **AndroidManifest.xml** file looks like.

```
cat AndroidManifest.xml

<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest
xmlns:android="http://schemas.android.com/apk/res/android"
android: compileSdkVersion="31" android: compileSdkVersionCodename="12"
package="com.example.enumeration01" platformBuildVersionCode="31"
platformBuildVersionName="12">
    <uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Figure 4 AndroidManifest.xml

All three of the examples will be further analysed and explained in more depth in the next chapters.

Performing Static and Dynamic Analysis

Reverse-engineering is the act of dismantling an object to see how it works. Among other reasons, reverse-engineering can be used to do a security analysis [13]. Reverse engineering can be done statically, and dynamically.

Static Analysis

Static Analysis is the automated analysis of source code without executing the application. Static Analysis is often used to detect [14]:

- Security vulnerabilities.
- Performance issues.
- Non-compliance with standards.
- Use of out of date programming constructs.

In Android applications we can do this kind of analysis with the help of some tools. As we discussed on a previous chapter, an apk file contains a .dex file which is actually the executable file that contains all the .class files of the application.

In order to able to read the source code of an application, we essentially need to decompile the .class files that are included in the .dex file. To do this, first we are going to use a tool that converts the .dex file into .class files, and zipped them in a .jar file. This tool is called dex2jar.

Once we have this files converted and zipped in a .jar file, we can go on and load it on another tool called JADX. This tool is responsible for decompiling the .class files and make the source code readable for humans. The image bellow shows the decompiled **MainActivity.java** file.

```
package com.example.reverse01;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;

/* loaded from: Reverse01-dex2jar.jar:com/example/reverse01/MainActivity.class */
public class MainActivity extends AppCompatActivity {
    TextView txv1;

    /* JADX INFO: Access modifiers changed from: protected */
    @Override // androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity,
    public void onCreate(Bundle bundle) {
        onCreate(bundle);
        setContentView(R.layout.activity_main);
        this.txv1 = (TextView) findViewById(R.id.editTextPassword);
        ((Button) findViewById(R.id.btnLogin)).setOnClickListener(new View.OnClickListener()
            @Override // android.view.View.OnClickListener
            public void onClick(View view) {
                Intent intent = new Intent(MainActivity.this, LoginActivity.class);
                intent.putExtra("pass", MainActivity.this.txv1.getText().toString());
                MainActivity.this.startActivity(intent);
            }
        });
    }
}
```

Figure 5 JADX

This was a method that allows us to only read the source code of the application. There are other tools that not only allows to decompile and read the code, but also change the code and recompile the application, creating this way a new apk file ready to run. This tool is Apktool.

As we can see the source code is not that human readable, since it is shown in **smali** and not in a Java-like pseudocode as we saw in JADX. More about the assembly language **smali** is going to be discussed in the Challenges Walkthrough chapter.

Below is an image showing a snippet of **smali** code opened with **vim** editor.

```
42     iget-object p1, p1, Lcom/example/reverse02/MainActivity;->o:Landroid/widget/TextView;$
43 $
44     invoke-virtual {p1}, Landroid/widget/TextView;->getText()Ljava/lang/CharSequence;$
45 $
46     move-result-object p1$
47 $
48     invoke-interface {p1}, Ljava/lang/CharSequence;->toString()Ljava/lang/String;$
49 $
50     move-result-object p1$
51 $
52     iget-object v0, p0, Lcom/example/reverse02/MainActivity$a;->b:Lcom/example/reverse02/MainActivity;$
53 $
54     invoke-virtual {v0}, Lcom/example/reverse02/MainActivity;->stringFromJNI()Ljava/lang/String;$
55 $
56     move-result-object v0$
57 $
58     invoke-virtual {p1, v0}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z$
59 $
60     move-result p1$
61 $
62     if-eqz p1, :cond_0$
```

Figure 6 APKTool smali code

Dynamic Analysis

When the analysis is performed while the software is running, then it is known as Dynamic Analysis. Dynamic analysis methods can be used to identify security vulnerabilities as well, apart from debugging the applications.

Frida is a dynamic code instrumentation toolkit. It lets you inject scripts into the application and inspect and change running processes.

Frida is used to bypass many of the techniques developers use to secure the apps. Some examples include bypassing the login screen to authenticate without a password, or disabling SSL pinning to allow hackers to see all the network traffic between the app and the backend servers [15].

Also one can use Frida to bypass root checks or hooking native functions and their return values. Below is an image showing Frida capturing the return value of the function **NewStringUTF**, from the shared library **libreverse04.so**.

Figure 7 Firda Native Function Hooking

All the above cases and tools will be further analysed and explained in more depth in the next chapters.

Extracting an APK File

Having the application installed in a device, allows us to experiment and see how it works. However, in order to further examine the application, having the APK file is necessary.

Some apps come pre-installed on Android devices, while other apps can be downloaded from Google Play. Apps downloaded from Google Play are automatically installed on your device, while those downloaded from other sources must be installed manually.

Download APK Online

Typically, users never see APK files because Android handles app installation in the background via Google Play or another app distribution platform. However, many websites offer direct APK file downloads for Android users who want to install apps manually.

Some of the websites that one can download APK files are:

- APKCombo
- Uptodown
- APKPure
- APKMirror

Extracting the APK using Third-Party Tools

APK files can also be extracted from the device. Third-Party tools can be installed in Android devices in order to extract the APK files of an already installed application. **APK Export** is an application that automatically exports the APK file of another application directly from the Google Play store.

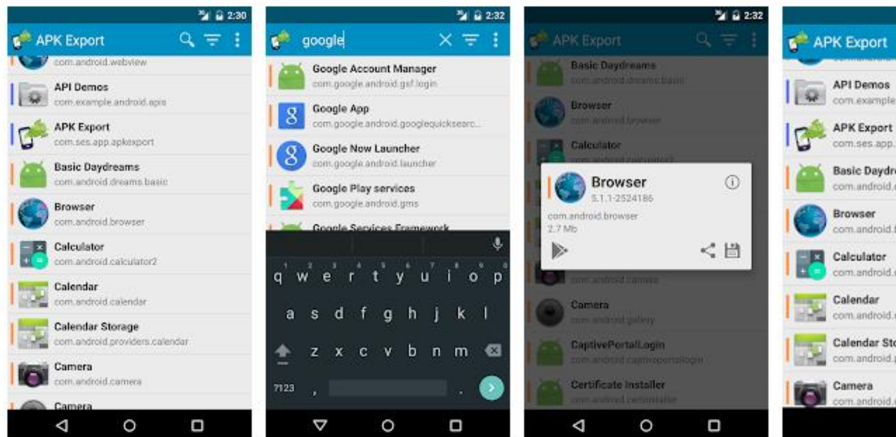


Figure 8 APK Extractor Third Party Tool

Once we have it installed, we can navigate through a screen that lists all the apps we have already installed from Google Play Store and choose the one we want to export the APK file. The exported file then, is stored locally in the device.

Extracting the APK from the Device

If an application is already installed in the device, the APK file is stored in the directory **/data/app/<package name>-1/base.apk**. For example, if the package name is **com.example.myapp**, the full path of the APK should be **/data/app/com.example.myapp-1/base.apk**.

We notice that the package name is followed by a number. In some android versions this is a sequence number, and in other versions it is a random string. Reading the content of the directory **/data/app/** is not permitted for non-root users and thus, it is difficult to guess the full package name of the app. To get the package name we can type the following command, since the app name is usually a part of the package name [16].

```
adb shell pm list packages | grep myapp
```

Figure 9 ADB grep app name

We notice the output of the above command contains the package name (com.example.myapp) of the app. Once we get the package name, we type the following command to get the full path of the APK file.

```
adb shell pm path com.example.myapp
```

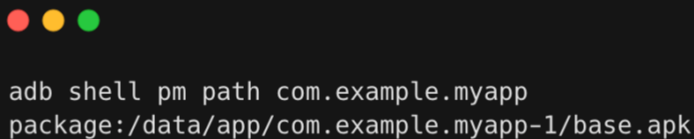


Figure 10 ADB pm path

Finally, we can retrieve the **base.apk** file, by typing the following command.

```
adb pull /data/app/com.example.myapp-1/base.apk
```

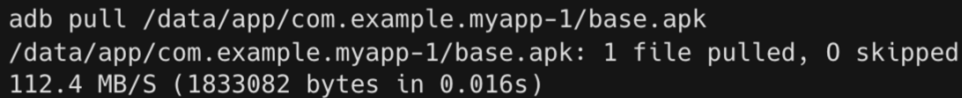


Figure 11 ADB pull apk

Capturing HTTP Requests

Often applications leak sensitive information in their network data, so finding it is one of the most crucial tasks of a penetration tester. The Insecure Communication, or Insufficient Transport Layer Protection, is the third biggest risk in mobile devices according to OWASP Mobile Top10 [17].

That could mean that users that are connected to a public network and they are using an application that submits their login credentials via HTTP to a server, sniffing attacks can successfully intercept their username and password.

Traffic Analysis is a Adynamic Analysis method. We could say that there are two types of Traffic Analysis. Passive and Active Analysis. In Passive Analysis we first capture all the network packets, and then we analyze them using a network analyzer, such as Wireshark.

In Active Analysis we actively intercept all the network traffic using a proxy server, and we read or modify the data on the fly. In the scenarios we are going to examine in the next chapters, we will be using only Active Traffic Analysis.

The tool that is going to be used for this type of attack is Burp Suite. More about this tool will been explained in the Challenges Walkthroughs chapter.

The image below shows an intercepted HTTP request using Burp, where the password parameter is captured in plaintext.

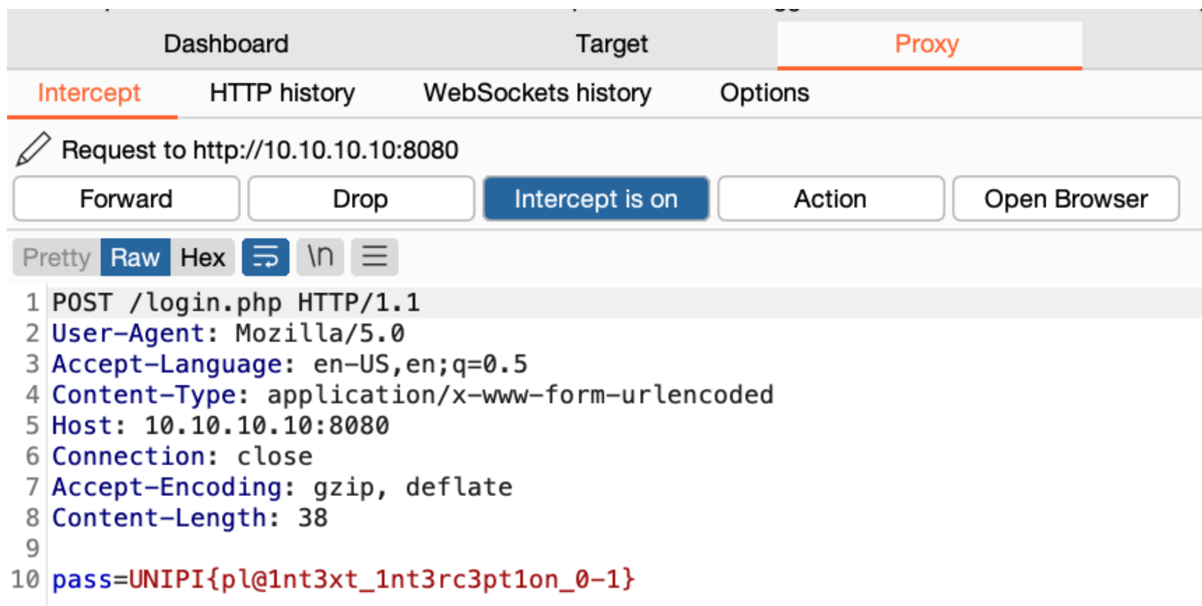


Figure 12 Burp Suit get request interception

Rooting the Device and Acquiring a Disk Image

With Android devices, it's possible to make a copy of the disk image in order to further examine it using tools like Autopsy. Autopsy is an open-source digital forensics platform that works well on Windows. This tool can help you retrieve deleted files and images from the disk, read databases, EXIF data, SMS and phone call logs, read the history of a web browser, and much else [18].

Rooting the device

In order to acquire an Android disk image, the device must be rooted. Rooting Android devices can be done in many ways. Having access to the device through the ADB terminal as user root, is not the same as having the device rooted. Rooting an Android Studio Virtual Device can be done by following the instructions provided in this GitHub project. This rooting technique has been tested in an AVD Nexus 5X. The following image shows the results of the application RootChecker, after successfully rooting the device [19].

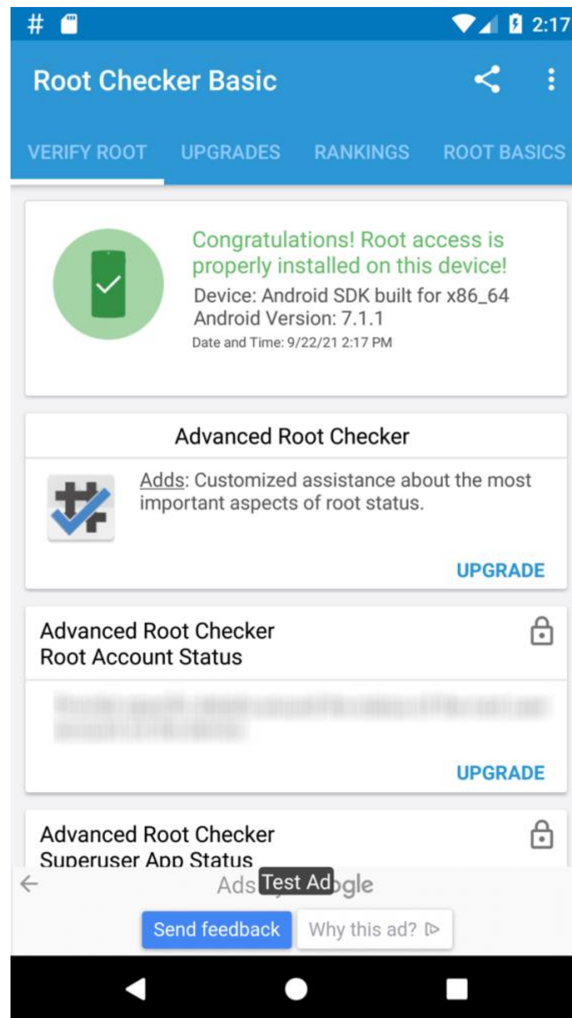


Figure 13 Root Checker app

Acquiring the disk image

Once the device is rooted, the disk image can be acquired by using the **dd** Unix utility and the BusyBox application. BusyBox is a software suite that provides several Unix utilities in a single executable file [20], and can be installed via ADB on a rooted device.

Assuming we have the device rooted and BusyBox is installed. Then, we need to find the partition that is mounted on the **/data** directory. The mounted partitions on an Android device can be listed like this.

```
adb shell mount
```

```
generic_x86_64:/ # mount
rootfs on / type rootfs (ro,seclabel,relatime)
tmpfs on /dev type tmpfs (rw,seclabel,nosuid,relatime,mode=755)
devpts on /dev/pts type devpts (rw,seclabel,relatime,mode=600)
proc on /proc type proc (rw,relatime,gid=3009,hidepid=2)
sysfs on /sys type sysfs (rw,seclabel,relatime)
selinuxfs on /sys/fs/selinux type selinuxfs (rw,relatime)
none on /dev/cpuctl type cgroup (rw,relatime,cpu)
/dev/block/vda on /system type ext4
(ro,seclabel,relatime,data=ordered)
/dev/block/vdb on /cache type ext4
(rw,seclabel,nosuid,nodev,noatime,errors=panic,data=ordered)
tmpfs on /storage type tmpfs (rw,seclabel,relatime,mode=755,gid=1000)
/dev/block/dm-0 on /data type ext4
(rw,seclabel,nosuid,nodev,noatime,errors=panic,data=ordered)
<SNIP>
```

Figure 14 List Mounted Drivers

Device drivers appear in the file system like normal files. The **dd** command-line utility is capable of backing up the boot sector of a hard drive [21]. Once we have found the right partition, we can use **dd** to acquire the disk image, and **nc** from the BusyBox app in order to send the disk image directly to our host machine.

```
adb shell "dd if=/dev/block/dm-0 | busybox nc -l -p 8888" &
```

Back to our host machine, we forward the traffic on port 8888 using ADB, and then we start a listener.

```
adb forward tcp:8888 tcp:8888
nc 127.0.0.1 8888 > disk.dd
```

```
~# adb shell "dd if=/dev/block/dm-0 | busybox nc -l -p 8888" &
[1] 5769

adb forward tcp: 8888 tcp: 8888 && nc 127.0.0.1 8888 > disk.dd
1638400+0 records in
1638400+0 records out
838860800 bvtes transferred in 27.893 secs (30074240 bytes/sec)
[1] + 5769 done

~# adb shell "dd if=/dev/block/dm-0 | busybox nc -1 -p 8888"

~# ls -1
total 1639384
-rW-r--r--
1 bertolis staff 838860800 Sep 16 16:10 disk.dd
```

Figure 15 Disk Image Acquisition

Once this is done, we can start Autopsy and follow the onscreen steps that provides, and import the exported disk image. The image below shows the main window of Autopsy while analysing an Android disk image.

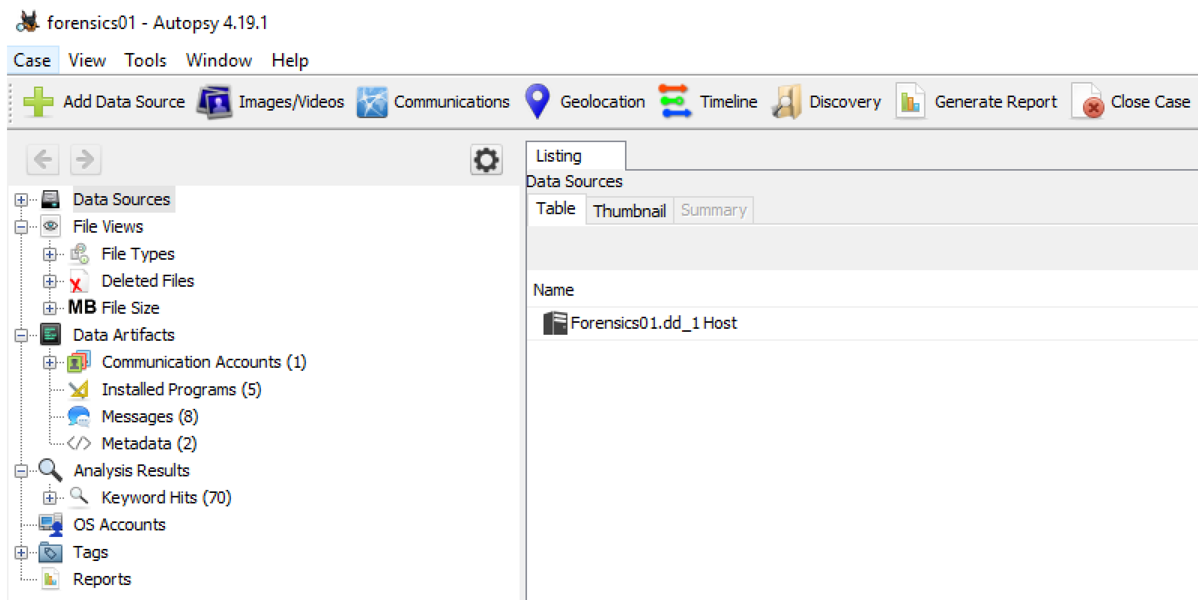


Figure 16 Autopsy Home Screen

Setting up the Environment

The environment that is going to be set up in this section, are tools and operating systems that are going to help us on assessing 10 Android challenges that I have already created. These are challenges of different subcategories, such as Enumeration, Reverse Engineering, Traffic Analysis and Forensics challenges.

These challenges are vulnerable applications, or applications that are developed implementing bad practices, resulting the lack of protection of their sensitive data.

The challenges were created using Android Studio IDE.

Android Emulator

Android application security assessment requires an Android device, in order to install, run and examine the application. In this documentation we are going to use an emulated device, but a real device can be used as well. The processes we will follow, will be the same for both alternatives.

There are several free emulators for Android on the internet. Some of them are:

- BlueStacks
  - Is known by many users to be the most comprehensive Android app player in the market. It is running on both Windows and Mac, and comes with a tone of features like Keymapping Tool, Instance Manager and Eco Mode, to improve the gamer's experience. It's also the safest emulator out there, with certified GDPR compliance.
- LDPlayer
  - Is a lightweight Android emulator focusing on gaming performance. Including good keyboard mapping controls, multi-instance, macros, high FPS, and graphical support.
- Android Studio

- Is the default development console for Android Studio IDE. It comes with a bunch of tools to help developers make apps and games specifically for Android. The setup is rather complicated so it won't appeal to everyone, but it is by far the fastest and most feature-rich option on this list.

- Bliss OS
  - It works as an Android emulator for PC via a virtual machine. However, it can also just flat run on your computer through a USB stick. As a VM install, the process is easy, but tedious if you've never made your own virtual machine before. The USB installation method is even more complicated, but it lets your computer actually run Android natively from boot.
- GameLoop
  - Is an Android emulator for gamers. This one is not good for productivity or developmental testing. However, this is a fairly decent gaming emulator that performs well with FPS games.
- Genymotion
  - This Android emulator is mostly for developers. You can configure the emulator for a variety of devices with various versions of Android to help suit your needs. It's not great for consumer uses. Its most useful feature is its availability on both your desktop computer and the cloud.
- MeMU
  - Is another excellent Android emulator that seems to do quite well with gamers, although it's usable as a productivity tool too. One of its biggest features is support for both AMD and Intel chipsets.

These were some of the best Android Emulators according to the Android Authority's latest publication [22].

Another Android Emulator that is worth to mention, is Corellium. Corellium is a paid emulator that provides online (cloud-based) virtualized iOS and Android devices. What is interesting about Corellium emulators, is the Arm-based virtualization that makes these virtual devices perform with native-like fidelity and speed, while also providing advanced capabilities that can't be replicated on a physical device [23].

Kernel exploitation is related to the CPU architecture, and most of the emulators virtualize a non-ARM CPU architecture like an x86 one. This makes it impossible for a pentester to work on a potential new kernel exploitation technique using a mobile emulator. Corellium seems to give the solution to this problem.

In this documentation we are going to use the emulator that Android Studio provides, since it includes many developer tools, offers a variety of devices and versions, and it is performing faster than the other free emulators.

Installing Android Studio on Linux is really easy. All we have to do is unzip it and run the file "studio.sh" inside the "bin/" directory. In order to install Android Studio on Windows or MacOS, all we have to do is click on the executable and follow the setup wizard. The process is pretty much the same for both the operating systems. After the installation has completed, we just need to wait for some components to be downloaded.

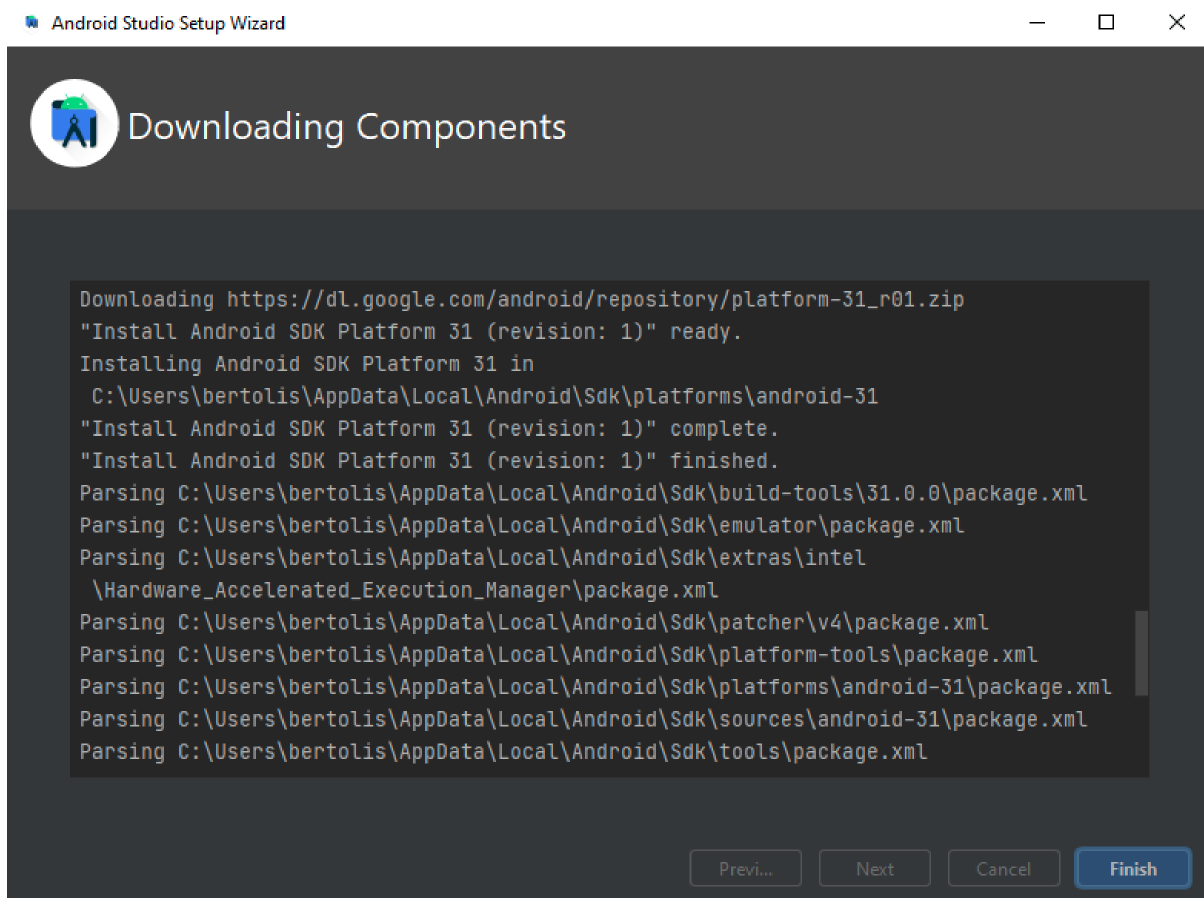


Figure 17 Android Studio Downloading Components

Once this is finished, click on **Finish** and in the next window click on **New Project**. Then select **Empty Activity** and click **Next**.

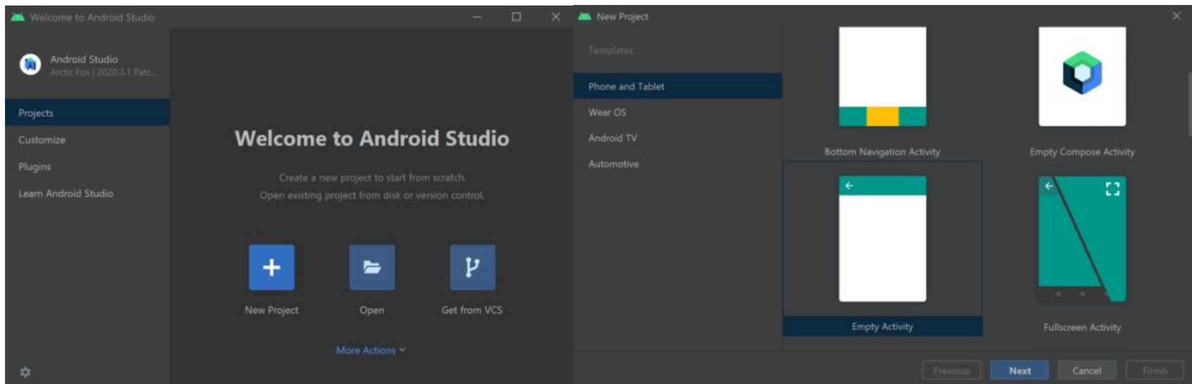


Figure 18 Android Studio New Project

Finally on the next screen, click finish to complete the process. Once the Android Studio start, we click on the drop-down menu on the top right of the window, and select **AVD Manager**.

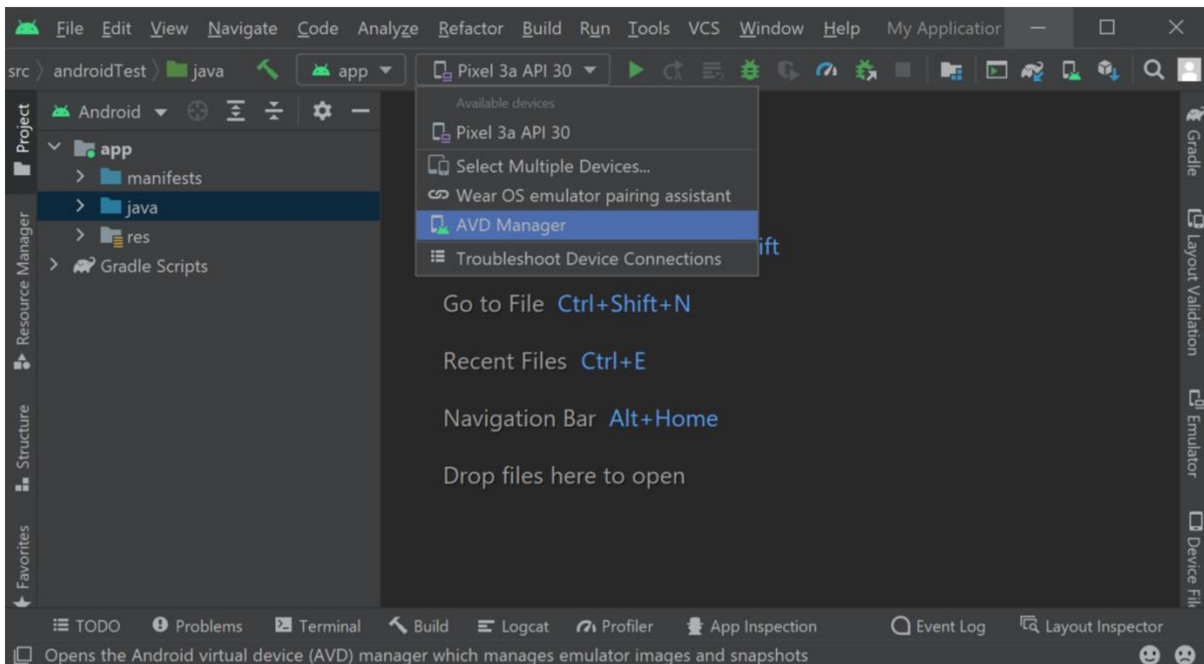


Figure 19 Android Studio AVD Manager

An [Android Virtual Device](#) (AVD) is a configuration that defines the characteristics of an Android phone that you want to simulate in the Android Emulator. The Device Manager is an interface you can launch from Android Studio that helps you create and manage AVDs [24].

On the AVD Manager window, click on the green "play" button to start the emulator.

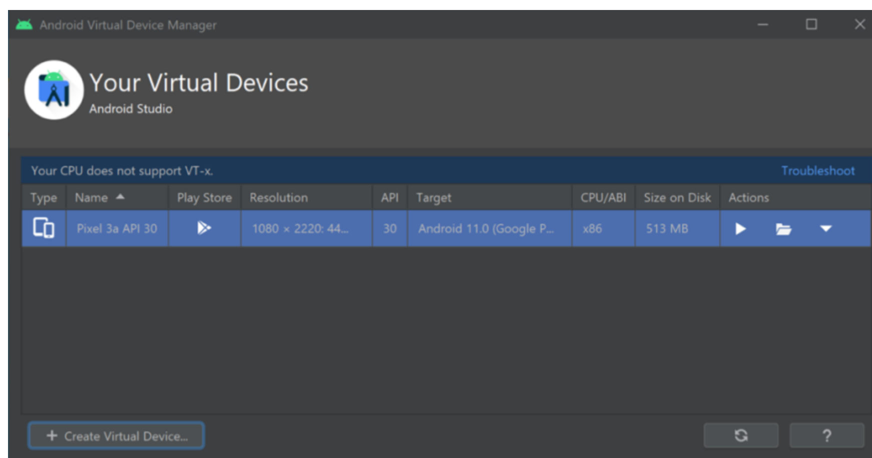


Figure 20 Android Studio Start Device

Once the device is started, it should look like this.

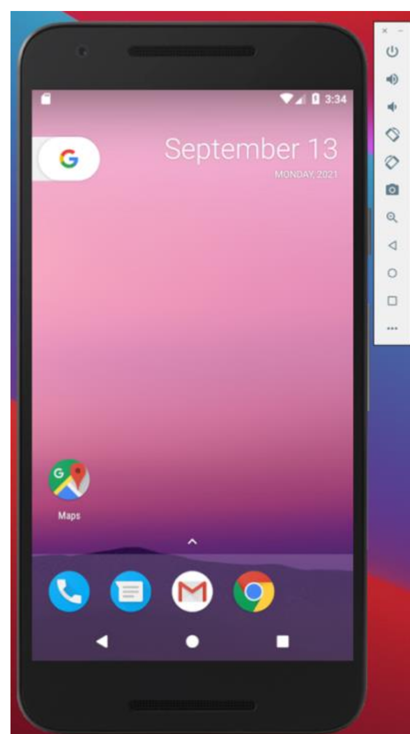


Figure 21 AVD Emulator

The Android Emulator is set up and ready for testing applications.

Android Debug Bridge

In order to be able to communicate with the emulated device we set up in the previous section, we will be using ADB (Android Debug Bridge). Android Debug Bridge (ADB) is a versatile command-line tool that lets you communicate with a device. The adb command facilitates a variety of device actions, such as installing and debugging apps, and it provides access to a Unix shell that you can use to run a variety of commands on a device. It is a client-server program that includes three components:

- **A client**, which sends commands. The client runs on your development machine. You can invoke a client from a command-line terminal by issuing an adb command.
- **A daemon** (adbd), which runs commands on a device. The daemon runs as a background process on each device.
- **A server**, which manages communication between the client and the daemon. The server runs as a background process on your development machine.

When you start an adb client, the client first checks whether there is an adb server process already running. If there isn't, it starts the server process. When the server starts, it is waiting for commands sent from the adb client. The server then sets up connections to all running devices. It locates emulators by scanning a range of default ports that are used by the emulators. Where the server finds an adb daemon (adbd), it sets up a connection to that port [25].

To install ADB on a Debian-based Linux we type the following.

```
sudo apt install adb
```

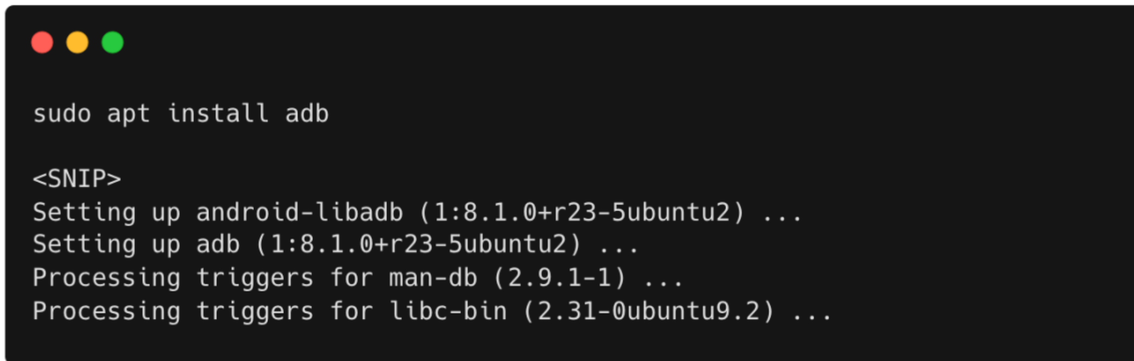


Figure 22 ADB Installation

In order to connect to the device via ADB, we need to enable **adb debugging** through the device's **Developer options**. We also need to connect the device to the same network with the host machine, so it can be found by the adb server which in turn, will establish the connections automatically.

Android Studio's virtual devices have already this option enabled, while they ensure that the device will be reachable from the adb server. That means that once the virtual device is started, the connection between the adb server and the device will be established automatically.

When a real device is used, first we have to connect to the same network. That can be achieved by opening the menu and navigate to the **Settings -> Network & internet -> Wi-Fi** screen, and select the network which your host machine is already connected to. Then click **Connect**.

The second thing we need to do, is to enable the **adb debugging** mode. To do so, we navigate to **Settings -> About device**, then we scroll down and tap 7 times on **Build number**, until we get the message "You are now a developer!". Then we navigate to **Settings -> System -> Advanced -> Developer options**, we scroll until we find and toggle on the **USB debugging** option.

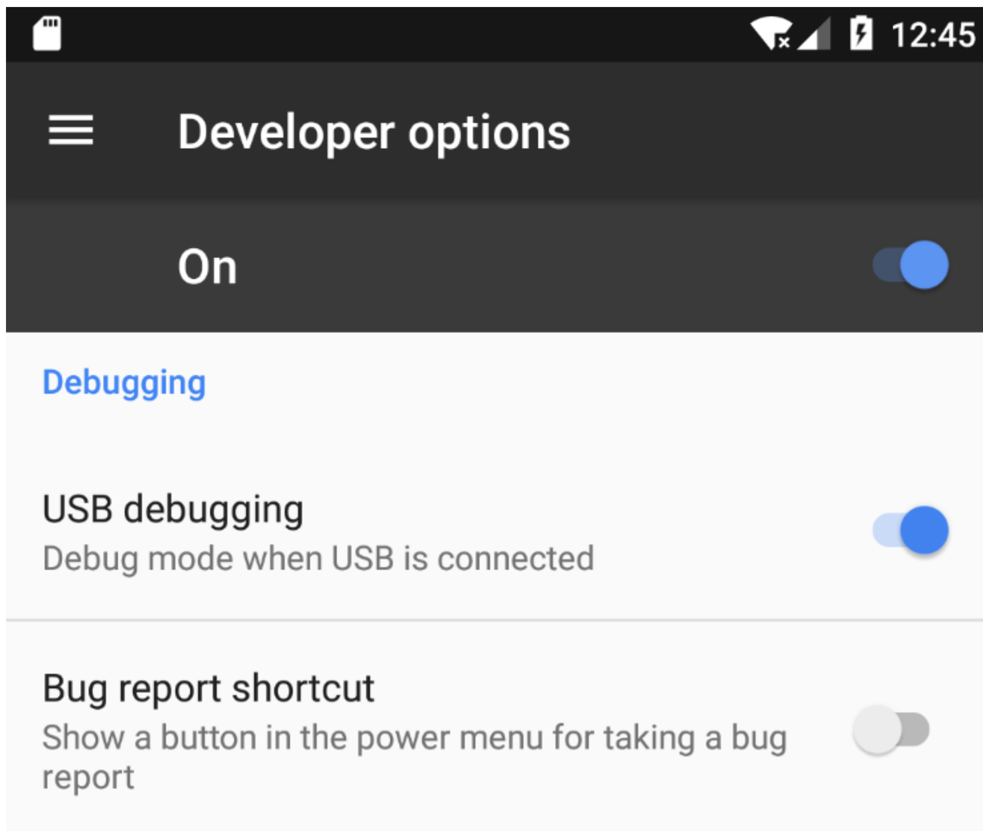


Figure 23 Enable USB Debagging

Once we are all set, we can check if the device is connected by issuing the following command.

```
adb devices
```

```
adb devices
List of devices attached
emulator-5554   device
```

Figure 24 ADB List Emulators

We can see that the device with the name **emulator-5554**, is connected to the adb server. Here are some useful adb commands [26].

- **adb help**: Llist all commands
- **adb devices**: Lists connected devices
- **adb root**: Restarts adbd with root permissions
- **adb kill-server**: Kills the adb server
- **adb install <apk>**: Install app
- **adb push <local> <remote>: C**opy file/dir to device
- **adb pull <remote> <local>**: Copy file/dir from device
- **adb logcat [options] [filter] [filter]**: View device log

Operating Systems and Tools

The main Operating System that will be used for conducting the assessment, is Parrot OS Security Edition. Parrot is a GNU/Linux distribution based on Debian and designed with Security and Privacy in mind. It includes a full portable laboratory for all kinds of cyber security operations, from pentesting to digital forensics and reverse engineering [27].

This is a lightweight operating system that contains many pre-installed security tools, that will help on conducting the Android application assessment. This OS will be used to assess most of the categories of the Android Security Lab.

Microsoft Windows will be also used in order to solve the challenges of the Forensics category. Microsoft Windows can be also downloaded for free, from Microsoft's official website.

Installation

Parrot OS is free and can be downloaded from their official website, allowing us to choose between the two different desktop environment, Mate and KDE.

Parrot OS can be easily installed on our physical or virtual machine, simply by following the officials installation guide that is provided on the website.

Microsoft also provide guides to follow, in order to install Windows on our machine.

Preparation

Regarding the security assessment part, some of the tools are already installed, and some others needs to be installed via the Apt package manager or downloaded from the internet.

Lab

- CTFd: Will allow us to host the challenges, and give access to the players
- Docker Engine: Will allow us to deploy the CTFd platform
- Docker Compose: Will allow us to configure in a single file all the services that CTFd is using, and start them all together using a single command

Assessment

- Android Studio: Will allows us to set up an Android virtual devise
- ADB: Will allow us to communicate with the Android virtual device
- sqlite3 client: Will allow us to enumerate the application's database
- Apktool: Will allow us to decompile, edit and recompile the APK files
- dex2jar: Will allow us to convert the apk to jar files, in order to import them in JADX
- JADX: Will allow us to decompile and read the source code of the apk files
- Ghidra: Will allow us to decompile and read the source code of the C++ files
- Frida: Will allow us to perform dynamic analysis and hook functions
- Burp Suite: Will allow us to intercept HTTP requests
- Autopsy: Will allow us to perform disk forensics

Setting up the CTF Platform

In order to provide the training content (challenges) in an efficient way, the CTFd platform is going to be used. CTFd is a Capture The Flag (CTF) framework designed for ease of use for both administrators and users. CTFd is an open source project, written in the simple Flask framework, and it's easy to customize with plugins and themes [28].

CTFd provides many features. Some of them are:

- Create your own challenges, categories, hints, and flags from the Admin Interface
- Individual and Team based competitions
- Scoreboard with automatic tie resolution
- Scoregraphs comparing the top 10 teams and team progress graphs
- Markdown content management system
- SMTP + Mailgun email support
  - Email confirmation support
  - Forgot password support
- Automatic competition starting and ending
- Team management, hiding, and banning
- Customize everything using the plugin and theme interfaces

CTFd can be set up in any operating system through pip (package manager for python), or **docker**. In this documentation, we are going to install CTFd through docker, as it allows us to experiment more and faster.

The operating system that will be used, is the Debian based Parrot Linux distribution. In order to host this platform, the only program that needs to be installed, is docker. Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers [29]. A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another [30].

Installation

Docker engine can be easily install using the **apt package manager**. From the terminal, we execute the following command.

```
sudo apt install docker.io
```

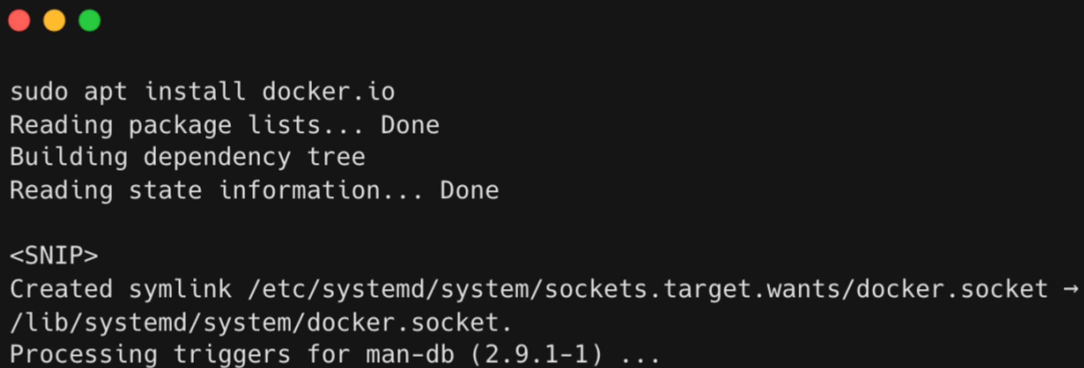


Figure 25 Docker Engin Installation

Once docker is installed, we also need to install docker-compose. Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration [31].

This tool allows us to configure all the different services that CTFd needs to run, in one file. Then we can start them by executing one command. Let's install docker-compose using the following commands.

```
sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

```
sudo curl -L
"https://github.com/docker/compose/releases/download/1.29.2/docker-
compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100   664  100   664    0     0   7460      0 --:--:-- --:--:-- --:--:--
7460
100 12.1M  100 12.1M    0     0  11.7M      0  0:00:01  0:00:01 --:--:--
22.2M
```

Figure 26 Docker Compose Installation

Then, we give the binary we just downloaded execution permissions, and create a symbolic link.

```
sudo chmod +x /usr/local/bin/docker-compose
sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose
docker-compose --version
```

```
docker-compose --version
docker-compose version 1.29.2, build 5becea4c
```

Figure 27 Docker Compose Version

Now that all the prerequisites are installed, let’s clone the CTFd project from their GitHub repository.

```
git clone https://github.com/CTFd/CTFd
```

```
git clone https://github.com/CTFd/CTFd
Cloning into 'CTFd'...
remote: Enumerating objects: 14119, done.
remote: Counting objects: 100% (1193/1193), done.
remote: Compressing objects: 100% (734/734), done.
remote: Total 14119 (delta 690), reused 826 (delta 432), pack-reused
12926
Receiving objects: 100% (14119/14119), 25.81 MiB | 10.33 MiB/s, done.
Resolving deltas: 100% (8754/8754), done.
```

Figure 28 Cloning CTFd Platform

Once it's cloned, we start all the services by executing the following command.

```
cd CTFd
sudo docker-compose up -d
```

```
sudo docker-compose up -d
Creating network "ctfd_internal" with the default driver
Creating network "ctfd_default" with the default driver
Pulling db (mariadb:10.4.12)...

<SNIP>
Status: Downloaded newer image for nginx:1.17
Pulling cache (redis:4)...

<SNIP>
Status: Downloaded newer image for redis:4
Creating ctfd_cache_1 ... done
Creating ctfd_db_1    ... done
Creating ctfd_ctfd_1  ... done
Creating ctfd_nginx_1 ... done
```

Figure 29 Starting CTFd Services

Since it is the first time we run this command, the docker images for each service that is used, must be downloaded first. A docker image is comparable to a snapshot in virtual machine

(VM) environments, and it contains application code, libraries, tools, dependencies and other files needed to make an application run [32].

As we can see on the image above, docker downloaded various images, such as **mariadb**, **nginx** and **redis**. Executing the following command, we can get information about the running containers.

```
docker ps -a
```

```
docker ps -a
CONTAINER ID   IMAGE            COMMAND                   CREATED
  STATUS         PORTS                                       NAMES
b2ad33d6be84   nginx:1.17       "nginx -g 'daemon of…"    8 minutes ago
  Up 8 minutes   0.0.0.0:80->80/tcp, :::80->80/tcp
ctfd_nginx_1
c8f2f6e92941   ctfd_ctfd        "/opt/CTFd/docker-en…"    8 minutes ago
  Up 8 minutes   0.0.0.0:8000->8000/tcp, :::8000->8000/tcp   ctfd_ctfd_1
0bfb62740219   mariadb:10.4.12  "docker-entrypoint.s…"    8 minutes ago
  Up 8 minutes                                               ctfd_db_1
3866f55a497b   redis:4          "docker-entrypoint.s…"    8 minutes ago
  Up 8 minutes
ctfd_cache_1
```

Figure 30 Docker Listing Services

As we can see, the **ctfd** service runs locally on port 8000. Let's open a web browser and try to reach the URL **http://127.0.0.1:8000**.

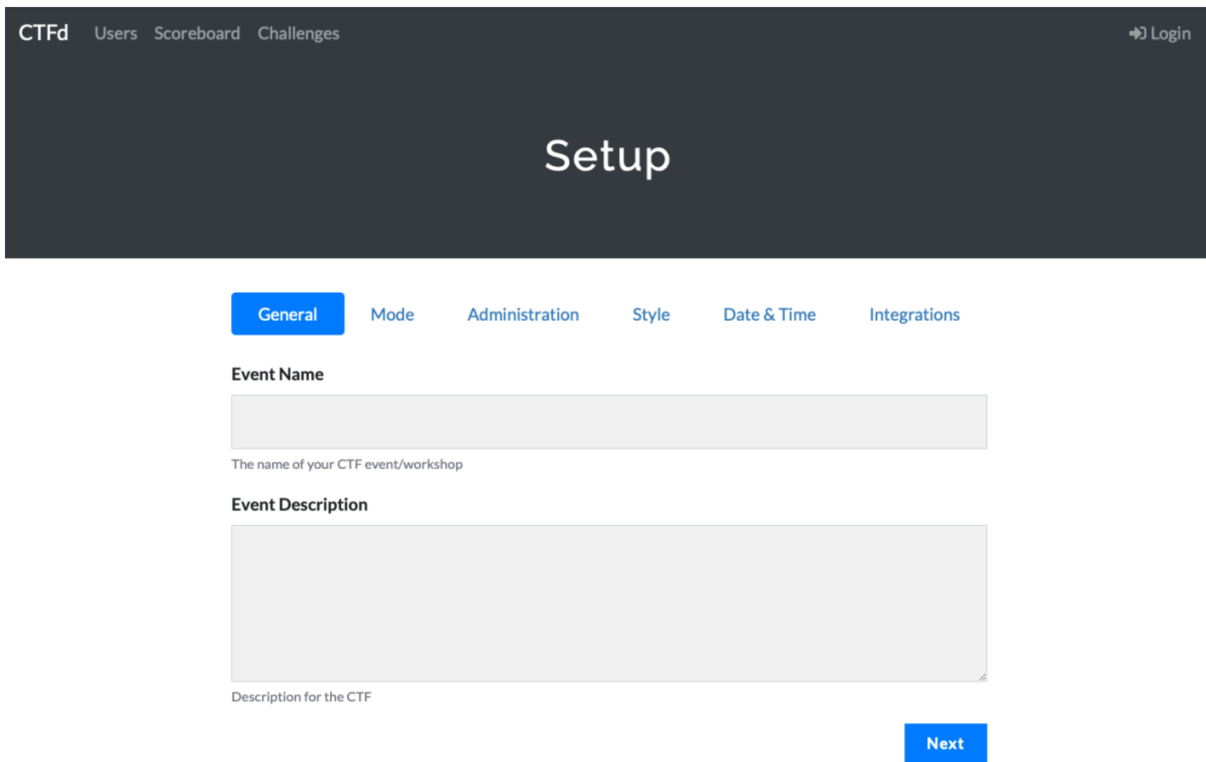


Figure 31 CTFd Suet Up Page

There are several steps one can follow in order to set up and customize the CTF event according to their needs. For the purpose of this documentation, I have already configured and customize the web app to look like an **Android Security Lab**.

A Lab that allows students to practice their skills on common Android security issues.

Figure 32 Android Security Training Lab Home Page

This is the home page of the **Android Security Lab**, after creating an **admin** account and login. Let's see now how we can upload and setup the challenges that I have already created locally. All the challenges that are going to be uploaded to this platform, are static. In other words, there will be a downloadable file for the player to download and examine locally. However, CTFd supports plugins for hosting dynamic challenges, such as ctfd-naumachia-plugin. Naumachia is a multi-tenant network sandbox for security challenges [33], and the whole project can be found here.

Back to our instance, from the top menu bar, we click on the **Admin Panel** tab.

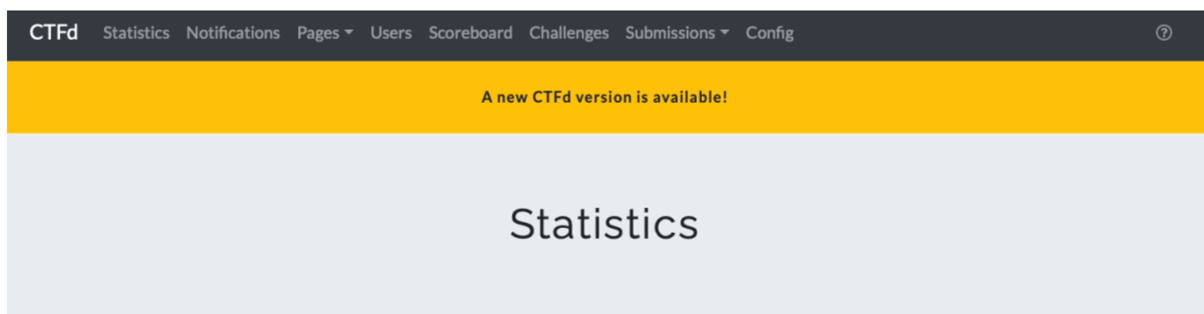


Figure 33 CTFd Admin Panel

We can see that a new menu bar is now available, providing more configuration options. Let's navigate to the **Challenges** tab and start adding challenges.

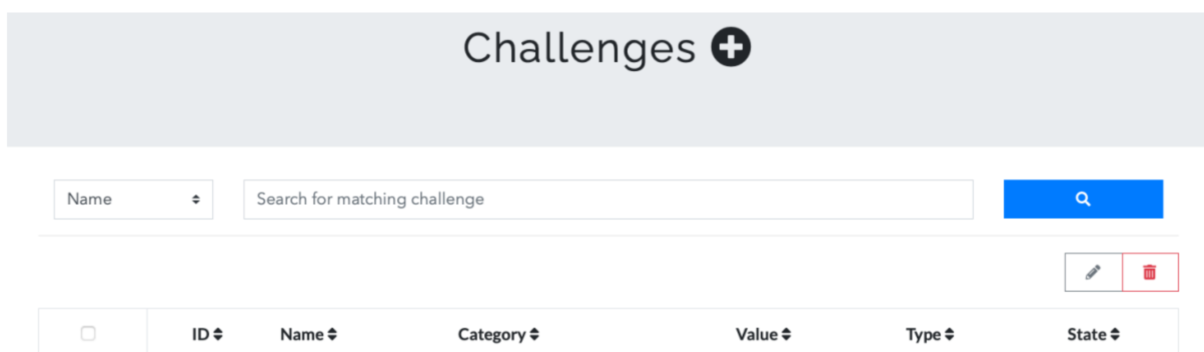


Figure 34 CTFd Challenges

Clicking on the plus symbol near the tittle **Challenges**, we are transferred to the web page from which we can start adding information regarding the challenge. Such information is the

type of the challenge, the **tittle**, **category**, **description** and **points** to award the player on solving the challenge.

Figure 35 CTFd Create New Challenge

Once we click on **Create** button, a window pops up asking to fill in some more options about the challenge, like the **flag**, the downloadable **file**, and the **state** of the challenge.

Figure 36 CTFd Challenges Properties

We can follow the same procedure for all of our challenges. Once we are done, we can click on the **CTFd** tab on the top left of the menu bar in order to switch from **admin** to **user** mode. Then, we navigate to the **Challenges** tab and check if all challenges are visible. The page should look like this.

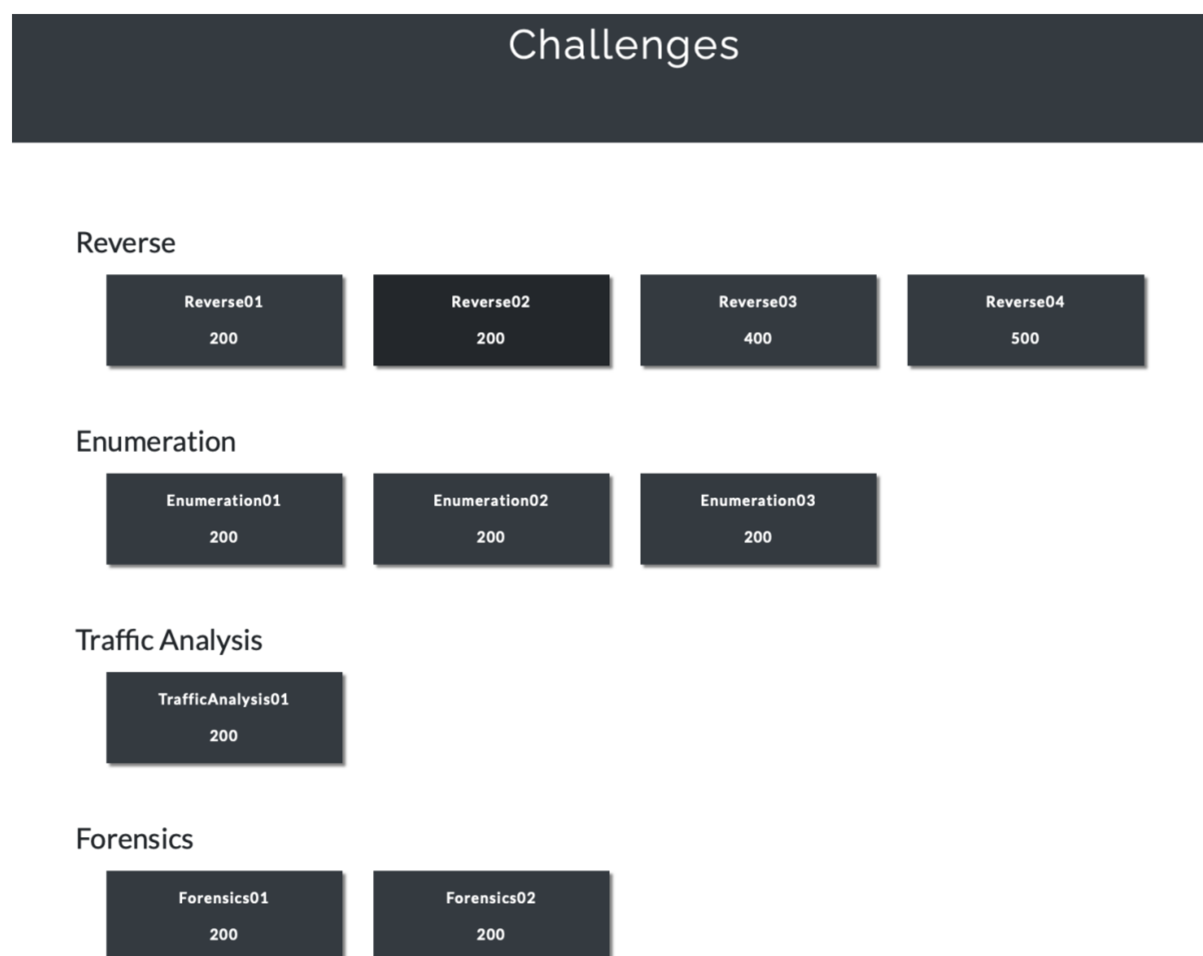


Figure 37 CTFd Download Challenges

We have successfully completed setting up the CTFd platform.

Challenges Walkthroughs

As discussed in a previous chapter, when someone solves a CTF challenge, a well-documented writeup must be provided as well. Apart from that, when the creator of the challenge is releasing one, the official writeup must be provided as well.

An official challenge writeup, or walkthrough, should describe in detail all the technologies and methods that are encountered when someone is resolving them.

The following are the official walkthroughs of the Android Security Training Lab project that were developed for the purposes of this paper. By following these walkthroughs, someone should be able to successfully complete this security training lab.

Enumeration

In this type of challenges, the player must collect information about the application. Then, they must reverse engineer it and try to find bad security practices. A flag, a specific predefined word of the format UNIPI{s0m3_r4nd0m_t3xt!}, will be indicating this bad practice or the results of it.

Enumeration01

Objective

The objective of this scenario is to show the basics on reverse engineering an apk file, and enumerate the common directories and files that contain critical information, like the Manifest.xml file.

Description

An intern mobile developer created a password manager application. Although he is confident about the functionality of the application, he has some concerns regarding the security part. Can you help him assess this Android app, and spot any bad practises?

Difficulty: Easy

Flag: UNIPI{th3_e4sy_w4y_sucks}

Release: 1e55ac4e97ee7445aa4aec6b403f10ea160727177318d82990708b7be0c6fe32

Challenge

Start the Android Emulator that we have already set up in the previous chapter, and make sure that the emulator is attached to the ADB, by executing the following command.

```
adb devices
```

```
adb devices
List of devices attached
emulator-5554   device
```

Figure 38 Enumeration01 ADB List Devices

Once we have confirmed the above step, unzip the **Enumeration01.apk.zip** file and install the extracted file **Enumeration01.apk** on the device, by issuing the following commands.

```
unzip Enumeration01.apk.zip
adb install Enumeration01.apk
```

```
unzip Enumeration01.apk.zip
Archive:  Enumeration01.apk.zip
  inflating: Enumeration01.apk

adb install Enumeration01.apk
Performing Streamed Install
Success
```

Figure 39 Enumeration01 Unzip APK

Then, we can find the installed application in the device's menu, and tap on to start it.

Password Manager

Please enter your master
password to login to the app.

Master password

LOGIN

Figure 40 Enumeration01 App

This is a password manager application. As the description implies, we are tasked to spot bad practises in this application. Let's start by decompiling the apk file. To do this, we will use the tool called Apktool. Apktool, is a tool for reverse engineering 3rd party, closed, binary Android apps. It can decode resources to nearly original form and rebuild them after making some modifications [34].

By decompiling the apk file, apart from the source code, we will have access to other configuration files as well. To download it, run the following command.

```
wget https://bitbucket.org/iBotPeaches/apktool/downloads/apktool_2.6.0.jar
```

```
Resolving bbuseruploads.s3.amazonaws.com
(bbuseruploads.s3.amazonaws.com)... 54.231.129.145
Connecting to bbuseruploads.s3.amazonaws.com
(bbuseruploads.s3.amazonaws.com)|54.231.129.145|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 19964258 (19M) [application/x-java-archive]
Saving to: 'apktool_2.6.0.jar'

apktool_2.6.0.jar                100%
[=======================================================>]  19,04M
1,42MB/s    in 30s
```

Figure 41 Enumeration01 Download APKTool

Then, issue the following command to decompile the apk file.

```
java -jar apktool_2.5.0.jar d Enumeration01.apk
```

```
java -jar apktool_2.5.0.jar d Enumeration01.apk
I: Using Apktool 2.5.0 on Enumeration01.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file:
/home/bertolis/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Figure 42 Enumeration01 Decompiling APK

In case that Java is not installed, the following message will be displayed.

```
java -jar apktool_2.6.0.jar Enumeration01.apk

Command 'java' not found, but can be installed with:

apt install openjdk-11-jre-headless  # version 11.0.13+8-0ubuntu1~20.04,
or
apt install default-jre              # version 2:1.11-72
apt install openjdk-13-jre-headless  # version 13.0.7+5-0ubuntu1~20.04
apt install openjdk-16-jre-headless  # version 16.0.1+9-1~20.04
apt install openjdk-17-jre-headless  # version 17.0.1+12-1~20.04
apt install openjdk-8-jre-headless   # version 8u312-b07-0ubuntu1~20.04
```

Figure 43 Enumeration01 APKTool Error

In order to install Java, issue the following command.

```
sudo apt install default-jdk
```

Once the decompilation is successful, the directory **Enumeration01** will be created. Listing the content of this directory, reveals the following file structure.

```
ls -l Enumeration01
total 8
-rw-r--r-- 1 bertolis bertolis 1663 Feb 21 17:33 AndroidManifest.xml
-rw-r--r-- 1 bertolis bertolis  595 Feb 21 17:34 apktool.yml
drwxr-xr-x 1 bertolis bertolis  148 Feb 21 17:34 kotlin
drwxr-xr-x 1 bertolis bertolis   54 Feb 21 17:34 original
drwxr-xr-x 1 bertolis bertolis 3286 Feb 21 17:33 res
drwxr-xr-x 1 bertolis bertolis  392 Feb 21 17:34 smali
drwxr-xr-x 1 bertolis bertolis   34 Feb 21 17:34 unknown
```

Figure 44 Enumeration01 List Directory

Among other files and directories, the file **AndroidManifest.xml** has been extracted. The manifest file describes essential information about your app to the Android build tools, the Android operating system, and Google Play.

Among many other things as discussed in the Assessment Techniques chapter, the manifest file is required to declare the permissions that the app needs in order to access protected parts of the system or other apps. It also declares any permissions that other apps must have if they want to access the content of this app.

Let's go on and read the content of this file.

```
cat AndroidManifest.xml
```

```
cat AndroidManifest.xml
<?xml version="1.0" encoding="utf-8" standalone="no"?><manifest
xmlns:android="http://schemas.android.com/apk/res/android"
android:compileSdkVersion="31" android:compileSdkVersionCodename="12"
package="com.example.enumeration01" platformBuildVersionCode="31"
platformBuildVersionName="12">
    <uses-permission
android:name="android.permission.READ_EXTERNAL_STORAGE"/>
    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE
UNIPI{th3_e4sy_w4y_sucks}"/>
<SNIP>
```

Figure 45 Enumeration01 Flag

Among other things, we can see that the **WRITE_EXTERNAL_STORAGE** permissions have been given to this app. Assuming that an app targets API level 25 or lower, and it lists both READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE in its manifest. Then, if the app requests READ_EXTERNAL_STORAGE and the user grants it, the system also grants WRITE_EXTERNAL_STORAGE at the same time, because it belongs to the same STORAGE permission group and is also registered in the manifest [35].

This is a bad practice because the app can grant privileges to write in the external storage, without the user’s permission. The flag is also revealed at the same configuration line, indicating the implementation of the bad practice.

Flag: UNIPI{th3_e4sy_w4y_sucks}

Mitigations

According to this article, the best practice is to explicitly request every permission whenever it's needed.

Enumeration02

Objective

The objective of this scenario is to show the basics on enumerating the installation directory of an installed application. More specifically it shows how sensitive information can be retrieved from a database, if the data are not encrypted.

Description

A big mobile app development company has recently released a beta version of their new password manager application, and is looking for a team to conduct a security assessment. Their main concern is not to leak user's personal information. Can you ensure that this won't happen?

Difficulty: Easy

Flag: UNIPI{d0nt_f0rg3t_t0_3ncrypt}

Release: abe49e1fac61e952f606853bba5569a9d0ec2ab185f670a8147c5679ae3197bb

Challenge

Start the Android Emulator that we have already set up in the previous chapter, and make sure that the emulator is attached to the ADB, by executing the following command.

```
adb devices
```

```
adb devices
List of devices attached
emulator-5554   device
```

Figure 46 Enumeration02 ADB List Devices

Once we have confirmed the above step, unzip the **Enumeration02.apk.zip** file and install the extracted file **Enumeration02.apk** on the device, by issuing the following commands.

```
unzip Enumeration02.apk.zip
adb install Enumeration02.apk
```

```
unzip Enumeration02.apk.zip
Archive:  Enumeration02.apk.zip
  inflating: Enumeration01.apk

adb install Enumeration02.apk
Performing Streamed Install
Success
```

Figure 47 Enumeration02 Unzip APK

Then, we can find the installed application in the device's menu, and tap on to start it.

Password Manager

Please enter your master
password to login to the app.

Master password

LOGIN

Figure 48 Enumeration02 App

This is a password manager application. As the description implies, we are tasked to conduct a security assessment on this app, and make sure that sensitive data is not leaked.

Since we have the app installed, let’s take a look at the file structure of the application’s installation directory. Using the following commands, we can get a shell connection on the emulated device. Switching **adbd** to root mode, allows us to access the directory **/data/data**, which contains the installation directories of the installed applications.

```
adb root
adb shell
```

```
adb root
restarting adbd as root

generic_x86_64:/ # whoami
root
```

Figure 49 Enumeration02 ADB root

Once we have a shell connection on the device, and since the name of the app is **Enumeration02**, we can execute the following command to find the home directory of the app.

```
ls -l /data/data | grep enumeration02
```

```
generic_x86_64:/ # ls -l /data/data | grep enumeration02

drwx------  5 u0_a197        u0_a197        4096 2022-02-19 13:51
com.example.enumeration02
```

Figure 50 Enumeration02 grep App Name

Now that we know the name of the application's home directory, we can issue the following command to list its content.

```
ls -l /data/data/com.example.enumeration02
```

```
generic_x86_64:/ # ls -l /data/data/com.example.enumeration02

total 24
drwxrws--x 2 u0_a197 u0_a197_cache 4096 2022-02-19 13:32 cache
drwxrws--x 2 u0_a197 u0_a197_cache 4096 2022-02-19 13:32 code_cache
drwxrwx--x 2 u0_a197 u0_a197       4096 2022-02-19 13:51 databases
```

Figure 51 Enumeration02 Listing App Installation Directory

Along with the **cache** and **code_cache** directories, **database** directory is also revealed. Listing the contents of this directory reveals the database **appDatabase**.

```
ls -l /data/data/com.example.enumeration02/databases
```

```
generic_x86_64:/ # ls -l /data/data/com.example.enumeration02/databases

total 96
-rw-rw---- 1 u0_a197 u0_a197  4096 2022-02-19 13:51 appDatabase
-rw------- 1 u0_a197 u0_a197 32768 2022-02-19 13:51 appDatabase-shm
-rw------- 1 u0_a197 u0_a197 45352 2022-02-19 13:51 appDatabase-wal
```

Figure 52 Enumeration02 Database

Let's use the pre-installed **sqlite3** client on the emulated device, to read the content of the database. Using the following command, we can execute sql commands interactively.

```
sqlite3 /data/data/com.example.enumeration02/databases/appDatabase
```

```
sqlite3 /data/data/com.example.enumeration02/databases/appDatabase

SQLite version 3.22.0 2018-12-19 01:30:22
Enter ".help" for usage hints.
sqlite>
```

Figure 53 Enumeration02 SQLite3

While the **sqlite>** sign is displayed, we can interactively execute sql commands to the databases. Let's list the tables.

```
sqlite> .tables
```

```
sqlite> .tables
SiteCredentials    android_metadata   room_master_table
```

Figure 54 Enumeration02 Tables

Among some default tables, **SiteCredentials** is listed as well. Executing the following query, we can list the columns of this table, along with their data.

```
sqlite> select * from SiteCredentials
```

```
sqlite> select * from SiteCredentials;

1|http://passmanager.com|John|UNIPI{d0nt_f0rg3t_t0_3ncrypt}
```

Figure 55 Enumeration02 Flag

The username and password for the website **http://passmanager.com** that are stored in this password manager app, are not encrypted and anyone with access to this application can see them.

Flag: UNIPI{d0nt_f0rg3t_t0_3ncrypt}

Mitigations

Data that is stored inside the database, must be encrypted.

Enumeration03

Objective

The objective of this scenario is to show the basics on enumerating the installation directory of an installed application. More specifically, it shows how one can read files that store sensitive information.

Description

A student has been assigned to create a password manager app for his project. The instruction of the project emphasised that the user data should be secured and inaccessible to third party users. Can you assess this app and help this student get a good score?

Difficulty: Easy

Flag: UNIPI{d0nt_f0rg3t_t0_3ncrypt_v0lum3_tw0}

Release: 7c05ff01615eb9461368508550f79300ae908e3a93b348edf9ee884eb63bf101

Challenge

Start the Android Emulator that we have already set up in the previous chapter, and make sure that the emulator is attached to the ADB, by executing the following command.

```
adb devices
```

```
adb devices
List of devices attached
emulator-5554	device
```

Figure 56 Enumeration03 List Devices

Once we have confirmed the above step, unzip the **Enumeration03.apk.zip** file and install the extracted file **Enumeration03.apk** on the device, by issuing the following commands.

```
unzip Enumeration03.apk.zip
adb install Enumeration03.apk
```

```
unzip Enumeration02.apk.zip
Archive:  Enumeration03.apk.zip
  inflating: Enumeration01.apk

adb install Enumeration03.apk
Performing Streamed Install
Success
```

Figure 57 Enumeration03 Unzip APK

Then, we can find the installed application in the device's menu, and tap on to start it.

Password Manager

Please enter your master
password to login to the app.

Master password

LOGIN

Figure 58 Enumeration03 App

This is a password manager application. As the description implies, our main concern is to check if the user data that is stored in this application is secured and not accessible from third party users.

Since the app is installed, we can start enumerating the file structure of the application's installation directory. Use the following commands to get a shell connection on the emulated device. Then, switching **adbd** to root mode to allow access to the directory **/data/data**, which contains the installation directories of the installed applications.

```
adb root
adb shell
```

```
adb root
restarting adbd as root

generic_x86_64:/ # whoami
root
```

Figure 59 Enumeration03 ADB root

Once we have a shell connection on the device, and since the name of the app is **Enumeration03**, we can execute the following command to find the home directory of the app.

```
ls -l /data/data | grep enumeration03
```

```
generic_x86_64:/ # ls -l /data/data | grep enumeration03

drwx------  5 u0_a198        u0_a198        4096 2022-02-19 15:46
com.example.enumeration03
```

Figure 60 Enumeration03 grep App

Now that we know the name of the application's home directory, we can issue the following command to list its content.

```
ls -l /data/data/com.example.enumeration03
```

```
generic_x86_64:/ # ls -l /data/data/com.example.enumeration03

total 24
drwxrws--x 2 u0_a198 u0_a198_cache 4096 2022-02-19 15:46 cache
drwxrws--x 2 u0_a198 u0_a198_cache 4096 2022-02-19 15:46 code_cache
drwxrwx--x 2 u0_a198 u0_a198       4096 2022-02-19 15:46 shared_prefs
```

Figure 61 Enumeration03 shared_prefs Directory

Along with the **cache** and **code_cache** directories, **shared_prefs** directory is also revealed. In Android, SharedPreferences are APIs that are used to save a relatively small collection of key-value pairs [36]. Such values are stored in XML files. Listing the contents of the directory **shared_prefs,** reveals the file **Credentials.xml**.

```
ls -l /data/data/com.example.enumeration03/shared_prefs/
```

```
generic_x86_64:/ # ls -l
/data/data/com.example.enumeration03/shared_prefs/

total 8
-rw-rw---- 1 u0_a198 u0_a198 183 2022-02-19 15:46 Credentials.xml
```

Figure 62 Enumeration03 SharedPreferences.xml

Let's read the content of this file, by executing the following command.

```
cat /data/data/com.example.enumeration03/shared_prefs/Credentials.xml
```

```
generic_x86_64:/ # cat
/data/data/com.example.enumeration03/shared_prefs/Credentials.xml

<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="uname">John</string>
    <string name="pass">UNIPI{d0nt_f0rg3t_t0_3ncrypt_v0lum3_tw0}
</string>
</map>
```

Figure 63 Enumeration03 Flag

The key-value pair **uname/pass** is revealed in plaintext.

Flag: UNIPI{d0nt_f0rg3t_t0_3ncrypt_v0lum3_tw0}

Mitigations

According to the official Android documentation, shared preferences should be edited by calling the **edit()** method of the **EncryptedSharedPreferences** object, instead of the **SharedPreferences** object, so the data can be encrypted [37].

Reverse

In this type of challenges, the player should reverse engineer that application, and perform static and dynamic analysis in order to find bad security practices. A flag, a specific predefined word of the format UNIPI{s0m3_r4nd0m_t3xt!}, will be indicating this bad practice or the results of it.

Reverse01

Objective

The objective of this scenario is to learn how to reverse engineer an apk file, read the source code, and eventually extract sensitive information like passwords.

Description

A friend of mine is about to release a password manager app on the mobile store. He asked me to check if I can somehow find a way to login to the app unexpectedly. Can you help me test this app before he uploads it?

Difficulty: Easy

Flag: UNIPI{d0nt_f0rg3t_my_p4ssw0rd!}

Release: 3c5697ee5dea835db31aad463422ef100ada47042e054aef79567048724275d5

Challenge

Start the Android Emulator that we have already set up in the previous chapter, and make sure that the emulator is attached to the ADB, by executing the following command.

```
adb devices
```

```
adb devices
List of devices attached
emulator-5554	device
```

Figure 64 Reverse01 ADB List Devices

Once we have confirmed the above step, unzip the **Reverse01.apk.zip** file and install the extracted file **Reverse01.apk** on the device, by issuing the following commands.

```
unzip Reverse01.apk.zip
adb install Reverse01.apk
```

```
unzip Reverse01.apk.zip
Archive:  Reverse01.apk.zip
  inflating: Reverse01.apk

adb install Reverse01.apk
Performing Streamed Install
Success
```

Figure 65 Reverse01 Unzip APK

Then, we can find the installed application in the device's menu and tap on to start it.

Figure 66 Reverse01 App

This is a password manager application. As the description implies, we are tasked to find a way, different than the one expected, and login to this application. Let’s reverse the apk file and take a look at the source code. To do this, first we need to understand what an apk file is.

APK (Android Package) is the file format that Android uses to distribute and install apps. It is an archive file that contains multiple files along with their metadata [38]. Using a file archiver like the cli tool **unzip** on Linux operating systems, we can decompress the APK file, and get the files that are included. By decompressing the **Reverse01.apk** file, we get the following output.

```
unzip Reverse01.apk
ls -l
```

```
unzip Reverse01.apk
Archive:  Reverse01.apk
  inflating: META-INF/com/android/build/gradle/app-metadata.properties
  inflating: res/z1.xml
  inflating: res/yP.xml
 extracting: res/9X.9.png
<SNIP>

ls -l
total 14568
-rw-r--r-- 1 bertolis bertolis    3896 Jan  1  1981 AndroidManifest.xml
-rw-r--r-- 1 bertolis bertolis    1719 Jan  1  1981 DebugProbesKt.bin
drwxr-xr-x 1 bertolis bertolis    3598 Feb 21 13:07 META-INF
-rw-r--r-- 1 bertolis bertolis 4916773 Feb 21 13:07 Reverse01.apk
-rw-rw-rw- 1 bertolis bertolis 9294592 Jan  1  1981 classes.dex
drwxr-xr-x 1 bertolis bertolis     148 Feb 21 13:07 kotlin
drwxr-xr-x 1 bertolis bertolis      58 Feb 21 13:07 lib
drwxr-xr-x 1 bertolis bertolis    6412 Feb 21 13:07 res
-rw-r--r-- 1 bertolis bertolis  690868 Jan  1  1981 resources.arsc
```

Figure 67 Decompress APK File

One of the files that are included inside an APK file, is the one with the **.dex** extension. In our case, the file **classes.dex** has been revealed. DEX files (Dalvik Executable), are used to hold a set of class definitions and their associated adjunct data [39]. It essentially contains code that is executed by the Android Runtime (ART).

Now that we have an idea of what the APK file is, we can use the tool dex2jar. This tool will extract the **.class** files from the **classes.dex** file, and zip it as a **jar** file. A JAR (Java ARchive) file is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file for distribution [40]. Let's go ahead and issue the following command to convert the files.

```
sudo apt install dex2jar
d2j-dex2jar Reverse01.apk
ls -l
```

```
d2j-dex2jar Reverse01.apk
dex2jar Reverse01.apk -> ./Reverse01-dex2jar.jar

ls -l
total 32536
-rw-r--r-- 1 bertolis bertolis  9092171 Feb 21 13:37 Reverse01-
dex2jar.jar
-rw-r--r-- 1 bertolis bertolis  4916773 Feb 20 17:06 Reverse01.apk
```

Figure 68 Reverse01 DEX2JAR

As we can see, the **Reverse01-dex2jar.jar** file is created. Now, let’s use [JADX] to read the source code. This tool is a **Dex to Java decompiler**, that producing Java source code from Android Dex and Apk files [41]. Using this tool, we can import the **Reverse01-dex2jar.jar** file we just created and read the Java source code of the application. To install and start the tool, we execute the following commands, and once it’s started, we locate and import the **Reverse01-dex2jar.jar** file.

```
sudo apt install jadx
jdax-gui
```

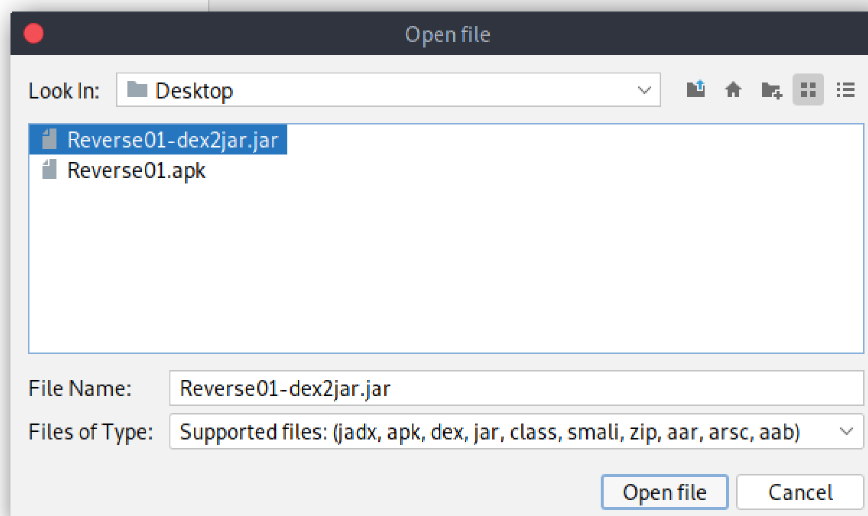


Figure 69 Reverse01 JADX

On the left side of the window, we can see a file structure similar to that in the Android Studio IDE. Navigating to **Source_code -> com -> example.reverse01**, we can spot and read the content on the **MainActivity** file.

```
File  View  Navigation  Tools  Help

Reverse01-dex2jar.jar
  Source code
    android.support.v4
    androidx
    com
      example.reverse01
        App
        AppDatabase
        AppDatabase_Impl
        BuildConfig
        CredentialsDao
        CredentialsDao_Impl
        InsertActivity
        ListAdapter
        LoginActivity
        MainActivity
        ManageActivity
        R
        Repository
        SiteCredentials
      google
    io.reactivex
    kotlin
    kotlinx.coroutines
    org
  Resources
  Summary
```

```
MainActivity
1  package com.example.reverse01;
2
3  import android.content.Intent;
4  import android.os.Bundle;
5  import android.view.View;
6  import android.widget.Button;
7  import android.widget.TextView;
8  import androidx.appcompat.app.AppCompatActivity;
9
10 /* loaded from: Reverse01-dex2jar.jar:com/example/reverse01/MainActivity.class */
11 public class MainActivity extends AppCompatActivity {
12     TextView txv1;
13
14     /* JADX INFO: Access modifiers changed from: protected */
15     @Override // androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity,
16     public void onCreate(Bundle bundle) {
17         onCreate(bundle);
18         setContentView(R.layout.activity_main);
19         this.txv1 = (TextView) findViewById(R.id.editTextPassword);
20         ((Button) findViewById(R.id.btnLogin)).setOnClickListener(new View.OnClickListener()
21             @Override // android.view.View.OnClickListener
22             public void onClick(View view) {
23                 Intent intent = new Intent(MainActivity.this, LoginActivity.class);
24                 intent.putExtra("pass", MainActivity.this.txv1.getText().toString());
25                 MainActivity.this.startActivity(intent);
26             }
27         });
28     }
29 }
```

Figure 70 Reverse01 MainActivity

Reading the source code, we notice that the value of edit text **editTextPassword** is stored into a variable called **txv1**. Then, when the **LoginActivity** is called, this variable is passed as the value of the parameter **pass**. That means that the user's input, which is probably the password, is passed and processed in the **LoginActivity**. Let's navigate to this activity and read its content.

```
MainActivity    LoginActivity
1  package com.example.reverse01;
2
3  import android.content.Intent;
4  import android.os.Bundle;
5  import android.widget.Toast;
6  import androidx.appcompat.app.AppCompatActivity;
7
8  /* loaded from: Reverse01-dex2jar.jar:com/example/reverse01/LoginActivity.class */
9  public class LoginActivity extends AppCompatActivity {
10     /* JADX INFO: Access modifiers changed from: protected */
11     @Override // androidx.fragment.app.FragmentActivity, androidx.activity.ComponentActivity,
12     public void onCreate(Bundle bundle) {
13         onCreate(bundle);
14         setContentView(R.layout.activity_main);
15         if (getIntent().getStringExtra("pass").equals("d261dc23942e81b5dcf8cb63f48949e1")) {
16             startActivity(new Intent(this, ManageActivity.class));
17             return;
18         }
19         Toast.makeText(this, "Wrong password!", 1).show();
20         startActivity(new Intent(this, MainActivity.class));
21     }
22 }
```

Figure 71 Reverse01 LoginActivity

Reading the source code of the file, we notice that password entered by the user, is being compared with the string **d261dc23942e81b5dcf8cb63f48949e1**, which is probably the login password. Let's go back to the application and try to login using this password.

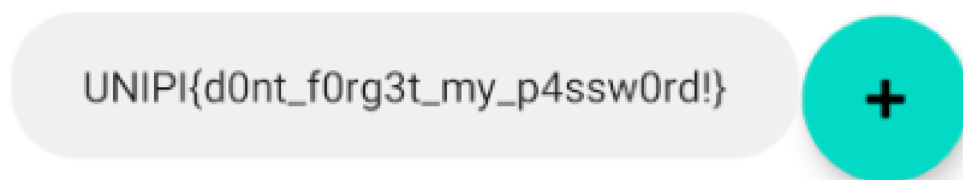


Figure 72 Reverse01 Flag

This is successful.

Flag: UNIPI{d0nt_f0rg3t_my_p4ssw0rd!}

Mitigations

In order to prevent the source code from such techniques, obfuscation methods can be used. Android official documentation provides an option while building your application, that among others, it allows us to obfuscate the source code, which results in changing and shorten the name of classes and members [42].

In order to enable this feature, we set the value **minifyEnabled** to **true**, inside the **build.gradle** file. The content of the file should look like this.

```
buildTypes {
    release {
        minifyEnabled true
        proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
    }
}
```

Figure 73 Reverse01 Mitigation

Reverse02

Objective

The objective of this scenario is to learn how to decompile, change the source code, and recompile an APK file, in order to bypass the login mechanism without knowing the password.

Description

The security company I have applied for, has put me through a test. I need to bypass the login screen of this app. Can you help me?

Difficulty: Easy

Flag: UNIPI{l0c@l_l0g1n_v@l1d@ti0n_suck5}

Release: 22059585251514e50d61870ce684dc7c02c48278949741271f8c4fdd3cc282be

Challenge

Start the Android Emulator that we have already set up in the previous chapter, and make sure that the emulator is attached to the ADB, by executing the following command.

```
adb devices
```

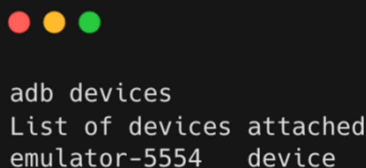


Figure 74 Reverse02 ABD List Devices

Once we have confirmed the above step, unzip the **Reverse02.apk.zip** file and install the extracted file **Reverse02.apk** on the device, by issuing the following commands.

```
unzip Reverse02.apk.zip
adb install Reverse02.apk
```

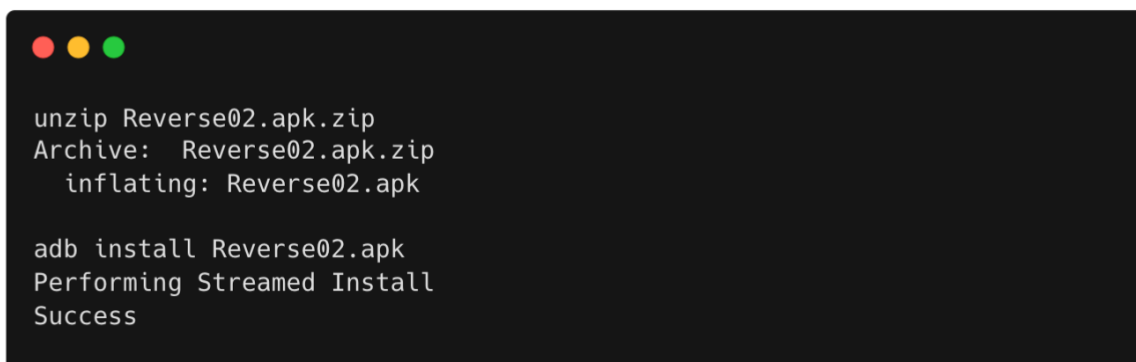


Figure 75 Reverse02 Unzip APK

Then, we can find the installed application in the device's menu and tap on to start it.

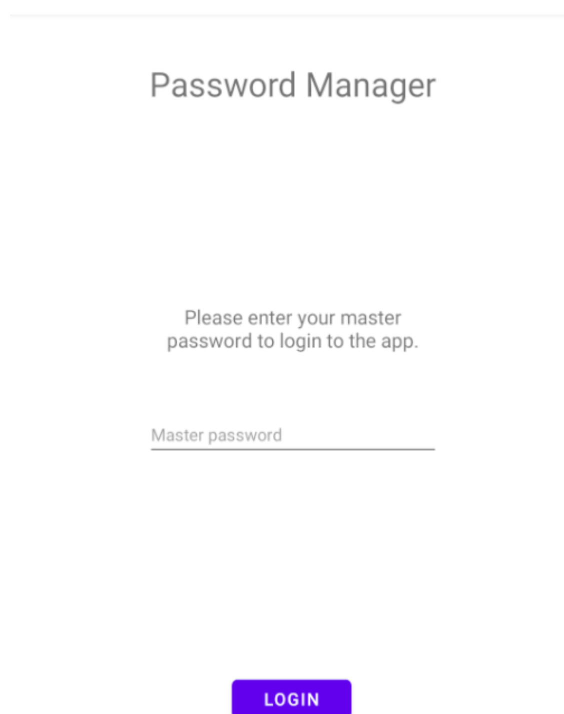


Figure 76 Reverse02 App

This is a password manager application. As the description implies, we are tasked to bypass the login screen of this application. Let's reverse the APK file and take a look at the source code.

Reversing the file using **DEX2JAR** and **JADX** like in the Reverse01 challenge, doesn't work. The string that is possibly going to be compared with the password entered by the user, isn't in plaintext. Instead, it is returned by a function called **stringFromJNI()**.

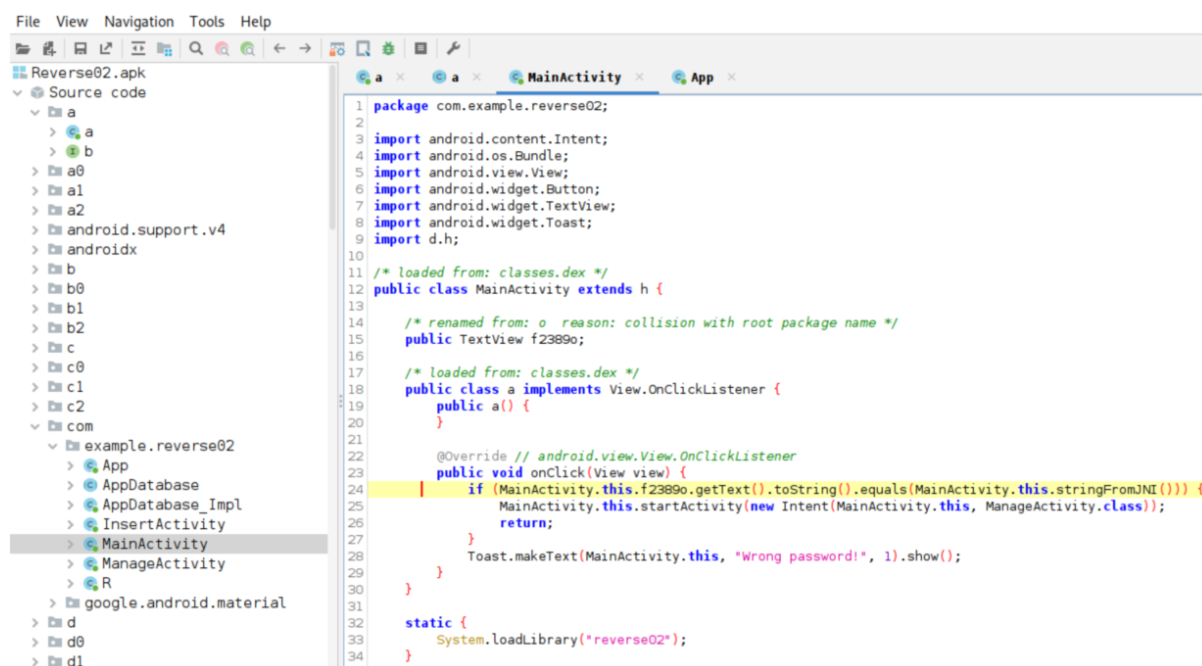


Figure 77 Reverse02 MainActivity

The source code also seems to be obfuscated, as we noticed that many methods and classes names are changed to random characters.

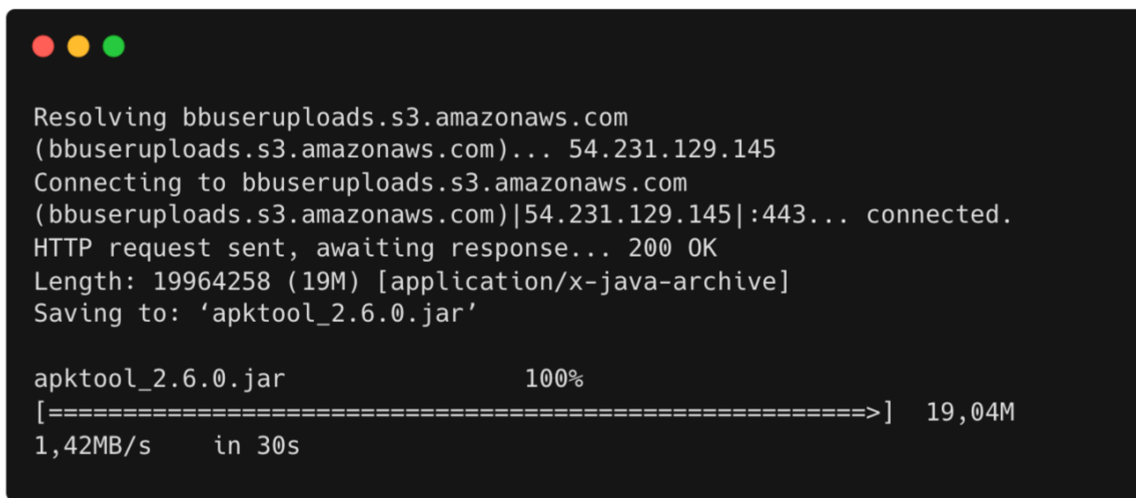
We can further examine this APK file by using the same tool we used on the **Enumeration01** challenge, Apktool. This tool, not only will extract configuration files from the **.apk** file, but it will convert a **.dex** file into a **.smali** file.

As we said earlier in the Reverse01 challenge, a **DEX** file (Dalvik Executable), is a file that contain the compiled Java code (Java classes) and can be interpreted by the Dalvik Virtual Machine. On the other hand, **Smali** is the assembly language used by the Android Dalvik

Virtual Machine [43]. Apktool, allows us to edit the converted **Smali** files, recompile them and then create a new APK file.

Let’s start by downloading the Apktool.

```
wget https://bitbucket.org/iBotPeaches/apktool/downloads/apktool_2.6.0.jar
```



```
Resolving bbuseruploads.s3.amazonaws.com
(bbuseruploads.s3.amazonaws.com)... 54.231.129.145
Connecting to bbuseruploads.s3.amazonaws.com
(bbuseruploads.s3.amazonaws.com)|54.231.129.145|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 19964258 (19M) [application/x-java-archive]
Saving to: ‘apktool_2.6.0.jar’

apktool_2.6.0.jar                100%
[=======================================================>]  19,04M
1,42MB/s    in 30s
```

Figure 78 Reverse02 Download APKTool

Then, go ahead and decompile the APK file.

```
java -jar apktool_2.5.0.jar d Reverse02.apk
```

```
java -jar apktool apktool_2.5.0.jar d Reverse02.apk
I: Using Apktool 2.5.0 on Reverse02.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file:
/home/bertolis/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Figure 79 Reverse02 APKTool Decompile

Listing the content of the extracted directory, we can see that the following file structure.

```
ls -l Reverse02
```

```
ls -l Reverse02
total 8
-rw-r--r-- 1 bertolis bertolis 1663 Feb 21 17:33 AndroidManifest.xml
-rw-r--r-- 1 bertolis bertolis  595 Feb 21 17:34 apktool.yml
drwxr-xr-x 1 bertolis bertolis  148 Feb 21 17:34 kotlin
drwxr-xr-x 1 bertolis bertolis   58 Feb 21 17:34 lib
drwxr-xr-x 1 bertolis bertolis   54 Feb 21 17:34 original
drwxr-xr-x 1 bertolis bertolis 3286 Feb 21 17:33 res
drwxr-xr-x 1 bertolis bertolis  392 Feb 21 17:34 smali
drwxr-xr-x 1 bertolis bertolis   34 Feb 21 17:34 unknown
```

Figure 80 Reverse02 Listing Decompressed APK

We notice that the directory **smali** has been created. Listing the directory **smali/com/example/reverse02/**, reveals the following **.smali** files.

```
ls -l Reverse02/smali/com/example/reverse02/
```

```
ls -l Reverse02/smali/com/example/reverse02/
total 76
-rw-r--r-- 1 bertolis bertolis 21309 Feb 21 17:59  App.smali
-rw-r--r-- 1 bertolis bertolis   292 Feb 21 17:59  AppDatabase.smali
-rw-r--r-- 1 bertolis bertolis 13652 Feb 21 17:59
'AppDatabase_Impl$a.smali'
-rw-r--r-- 1 bertolis bertolis  4744 Feb 21 17:59
AppDatabase_Impl.smali
-rw-r--r-- 1 bertolis bertolis  3324 Feb 21 17:59  InsertActivity.smali
-rw-r--r-- 1 bertolis bertolis  2639 Feb 21 17:59 'MainActivity$a.smali'
-rw-r--r-- 1 bertolis bertolis  1545 Feb 21 17:59  MainActivity.smali
-rw-r--r-- 1 bertolis bertolis  1457 Feb 21 17:59
'ManageActivity$a.smali'
-rw-r--r-- 1 bertolis bertolis  4662 Feb 21 17:59  ManageActivity.smali
```

Figure 81 Reverse02 Smali Files

Let's open the **MainActivity$a.smali**, and try to locate the **if()** statement that we found earlier while enumerating with **JADX**.

```
vim Reverse02/smali/com/example/reverse02/MainActivity\$a.smali
```

```
42     iget-object p1, p1, Lcom/example/reverse02/MainActivity;->o:Landroid/widget/TextView;$
43 $
44     invoke-virtual {p1}, Landroid/widget/TextView;->getText()Ljava/lang/CharSequence;$
45 $
46     move-result-object p1$
47 $
48     invoke-interface {p1}, Ljava/lang/CharSequence;->toString()Ljava/lang/String;$
49 $
50     move-result-object p1$
51 $
52     iget-object v0, p0, Lcom/example/reverse02/MainActivity$a;->b:Lcom/example/reverse02/MainActivity;$
53 $
54     invoke-virtual {v0}, Lcom/example/reverse02/MainActivity;->stringFromJNI()Ljava/lang/String;$
55 $
56     move-result-object v0$
57 $
58     invoke-virtual {p1, v0}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z$
59 $
60     move-result p1$
61 $
62     if-eqz p1, :cond_0$
```

Figure 82 Reverse02 Smali Code

Searching for the word “if”, we are transferred to this snippet of code. By reading it, we can realize that this is the same **if()** statement with the one we found while enumerating the source code with **JADX**. Its seems that the string of the **TextView** is getting compared with the value that is returned from the **stringFromJNI()** function. Let’s go ahead and delete this line. By doing this, the user’s input validation won’t work, and we will jump directly to the code that is executed inside the **if()** statement. Once, we have the line deleted. We run the following to recompile the files.

```
java -jar apktool_2.5.0.jar b Reverse02
```

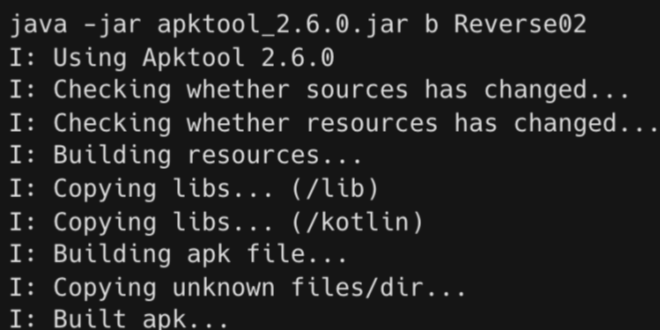


Figure 83 Reverse02 APKTool Recompile

Once it’s finished, we can find the new APK file in the following directory.

```
ls -l Reverse02/dist/
```

```
ls -l Reverse02/dist/
total 2480
-rw-r--r-- 1 bertolis bertolis 2539008 Feb 22 02:03 Reverse02.apk
```

Figure 84 Reverse02 New APK

What is left, is to sign the recompiled **Reverse02.apk** file using a self-signed certificate. To create a self-signed certificate, we are going to use **keytool**.

```
keytool -genkey -keystore john.keystore -validity 1000 -alias john
```

```
keytool -genkey -keystore john.keystore -validity 1000 -alias john
Enter keystore password:
Re-enter new password:

<SNIP>
What is your first and last name?
  [Unknown]:  john doe
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=john doe, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
correct?
  [no]:  Yes

Generating 2,048 bit DSA key pair and self-signed certificate
(SHA256withDSA) with a validity of 1,000 days
    for: CN=john doe, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown,
C=Unknown
```

Figure 85 Reverse02 Signing Certificate

Once the key is created, issue the following command to sign the APK file, using the password we entered during creating the certificate.

```
apksigner sign --ks john.keystore Reverse02/dist/Reverse02.apk
```

Figure 86 Reverse02 Signing The New APK

Now the APK is ready to be installed. Before installing the new APK file, make sure to delete the one that is already installed.

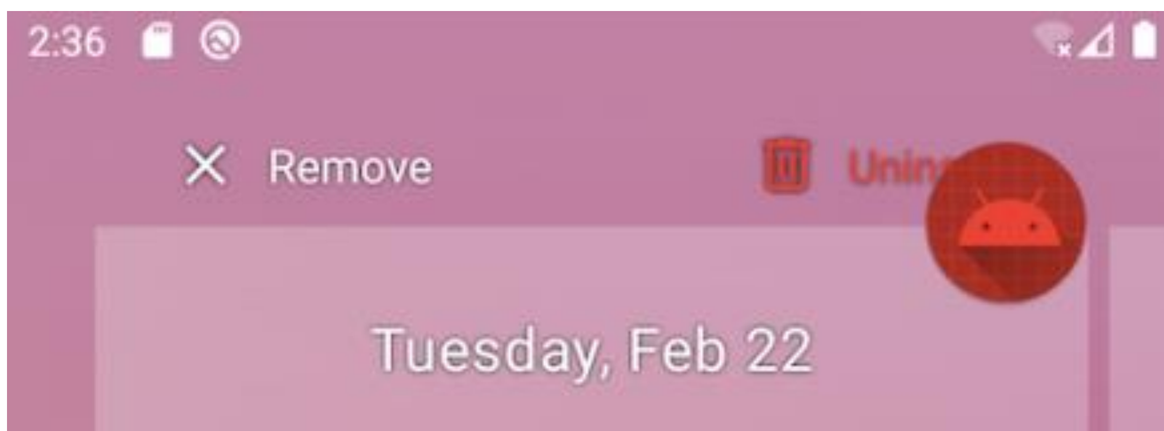


Figure 87 Reverse02 Uninstalling Old App

Then, we run the following to install the app.

```
adb install Reverse02/dist/Reverse02.apk
```

```
adb install Reverse02/dist/Reverse02.APK
Performing Streamed Install
Success
```

Figure 88 Reverse02 Installing New App

Finally, we locate and tap the app on the device to start. Once it starts, we click on the **LOGIN** button without entering any password, since we have removed the **if()** statement that validates the password.

Password Manager

http://passmanager.com
John
UNIPI{l0c@l_l0g1n_v@l1d@ti0n_suck5}

Figure 89 Reverse02 Flag

The login screen has been bypassed successfully.

Flag: UNIPI{l0c@l_l0g1n_v@l1d@ti0n_suck5}

Mitigations

Remote or encrypted authentication is resistant on such attacks, because there is no hardcoded password in plaintext in the app. As the App security best practices section in the Android official documentation suggests, enforcing secure communication improves app's stability and protects the data that are sent and received [44]. Combining these two practices strengthens the prevention of this kind of attacks.

Reverse03

Objective

The objective of this scenario is to learn how to reverse engineer an apk file, and search the source code in depth, in order to find sensitive information like passwords.

Description

I forgot the password I use to login to my mail account. Fortunately, I use the same password in my password manager app. Let's check if I can reverse the source code and get it.

Difficulty: Medium

Flag: UNIPI{n4t1v3_c0d3_1s_n0t_3n0ugh}

Release: dda0a4f41dbcb2a705e66be72583b0bd9f69f6fc375b59bbd87f64b662c54409

Challenge

Start the Android Emulator that we have already set up in the previous chapter, and make sure that the emulator is attached to the ADB, by executing the following command.

```
adb devices
```

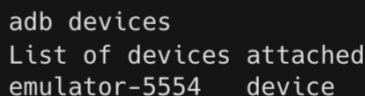


Figure 90 Reverse03 ADB Listing Devices

Once we have confirmed the above step, unzip the **Reverse03.apk.zip** file and install the extracted file **Reverse03.apk** on the device, by issuing the following commands.

```
unzip Reverse03.apk.zip
adb install Reverse03.apk
```

```
unzip Reverse03.apk.zip
Archive:  Reverse03.apk.zip
  inflating: Reverse03.apk

adb install Reverse03.apk
Performing Streamed Install
Success
```

Figure 91 Reverse03 Unzip APK

Then, we can find the installed application in the device's menu and tap on to start it.

Password Manager

Please enter your master
password to login to the app.

Master password

LOGIN

Figure 92 Reverse03 App

This is a password manager application. As the description implies, we are tasked to reverse the app and find the potential hardcode login password. Let’s reverse the APK file and take a look at the source code.

Reversing the file using **DEX2JAR** and **JADX** like in the Reverse01 challenge, gives us a first glance of the source code. The string that is possibly compared with the password that is entered by the user, isn’t in plaintext. Instead, it is returned by a function called **stringFromJNI()**.

```java
package com.example.reverse03;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.Toast;
import d.h;

/* loaded from: classes.dex */
public class MainActivity extends h {

    /* renamed from: o  reason: collision with root package name */
    public TextView f23890;

    /* loaded from: classes.dex */
    public class a implements View.OnClickListener {
        public a() {
        }

        @Override // android.view.View.OnClickListener
        public void onClick(View view) {
            if (MainActivity.this.f23890.getText().toString().equals(MainActivity.this.stringFromJNI())) {
                MainActivity.this.startActivity(new Intent(MainActivity.this, ManageActivity.class));
                return;
            }
            Toast.makeText(MainActivity.this, "Wrong password!", 1).show();
        }
    }
```

Figure 93 Reverse03 MainActivity

The source code also seems to be obfuscated as we noticed that many methods and classes names are changed to random characters. Searching online the **stringFromJNI()** function, reveals the following results.

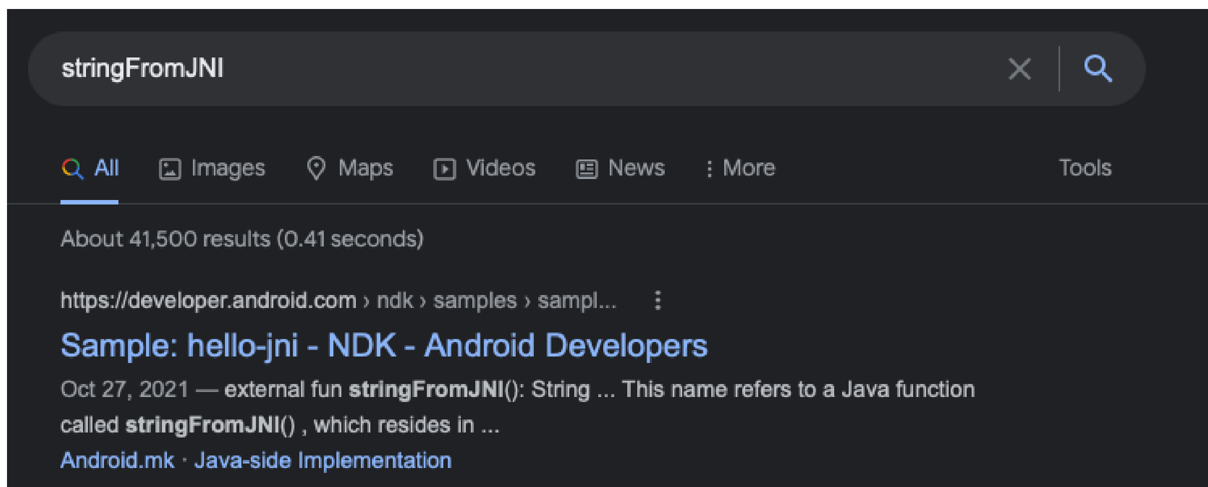


Figure 94 Reverse03 Google Search JNI

According to the Android official documentation, this is a function that is used when we create C/C++ application using the NDK. The NDK (Native Development Kit) is a set of tools that allows you to use C and C++ code with Android, and provides platform libraries you can use to manage native activities and access physical device components, such as sensors and touch input [45].

According to the documentation we saw earlier, the full name of the built library (the file that is created in C/C++) is libhello-jni.so, once the build system adds the lib prefix and the .so extension. Let's decompile **Reverse01.apk** using **Apktool** and search for a **.so** file.

```
wget https://bitbucket.org/iBotPeaches/apktool/downloads/apktool_2.6.0.jar
```

```
Resolving bbuseruploads.s3.amazonaws.com
(bbuseruploads.s3.amazonaws.com)... 54.231.129.145
Connecting to bbuseruploads.s3.amazonaws.com
(bbuseruploads.s3.amazonaws.com)|54.231.129.145|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 19964258 (19M) [application/x-java-archive]
Saving to: 'apktool_2.6.0.jar'

apktool_2.6.0.jar                100%
[=======================================================>]  19,04M
1,42MB/s    in 30s
```

Figure 95 Reverse03 APKTool Download

Once its downloaded, we execute the following command.

```
java -jar apktool_2.5.0.jar d Reverse03.apk
```

```
java -jar apktool_2.6.0.jar d Reverse03.apk
I: Using Apktool 2.6.0 on Reverse03.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file:
/home/bertolis/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Figure 96 Reverse03 Decompile APK

Enumeration of the **Reverse03/lib/x86_64/**, reveals the file **libreverse03.so**.

```
ls -l Reverse03/lib/x86_64/
total 224
-rw-r--r-- 1 bertolis bertolis 227656 Feb 22 15:59 libreverse03.so
```

Figure 97 Reverse03 Share Library

Reading the content of this file is not possible because it is compiled.

```
cat Reverse03/lib/x86_64/libreverse03.so

��!� ��uHux8�""�0�"�idr21e7075529GNU<�|
�=~�W�H�ꞷ�bZE�+����������P    W   k   ~   �    �
P�l8�N�
0�����q�!J"
����y�������"
��T �L(BP0
��4�N$
�!��M0rS�r�"5�t;PuA�p>��.�
p+�y�0y(�
P�!?o��g"�k@q���
7   t���N(}p�'�]��M�N(� s���d��ps���k��px��r<�n�
```

Figure 98 Reverse03 Shared Object Content

An SO file is a shared library used by programs installed on Linux and Android. Developers often build SO files using the "gcc" C/C++ compiler included in the GNU Compiler Collection (GCC) [46]. Let’s try to decompile the file **libreverseo3.so** using Ghidra.

Ghidra is a software reverse engineering (SRE) framework that includes a suite of full-featured high-end software analysis tools that, enable users to analyze compiled code. It also supports a wide variety of processor instruction sets and executable formats [47].

In order to install Ghidra we type the following.

```
sudo apt install ghidra
```

```
sudo apt install ghidra
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
<SNIP>
```

Figure 99 Reverse03 Ghidra Installation

Next, we start **ghidra** from the terminal, and create a new project.

```
ghidra
```

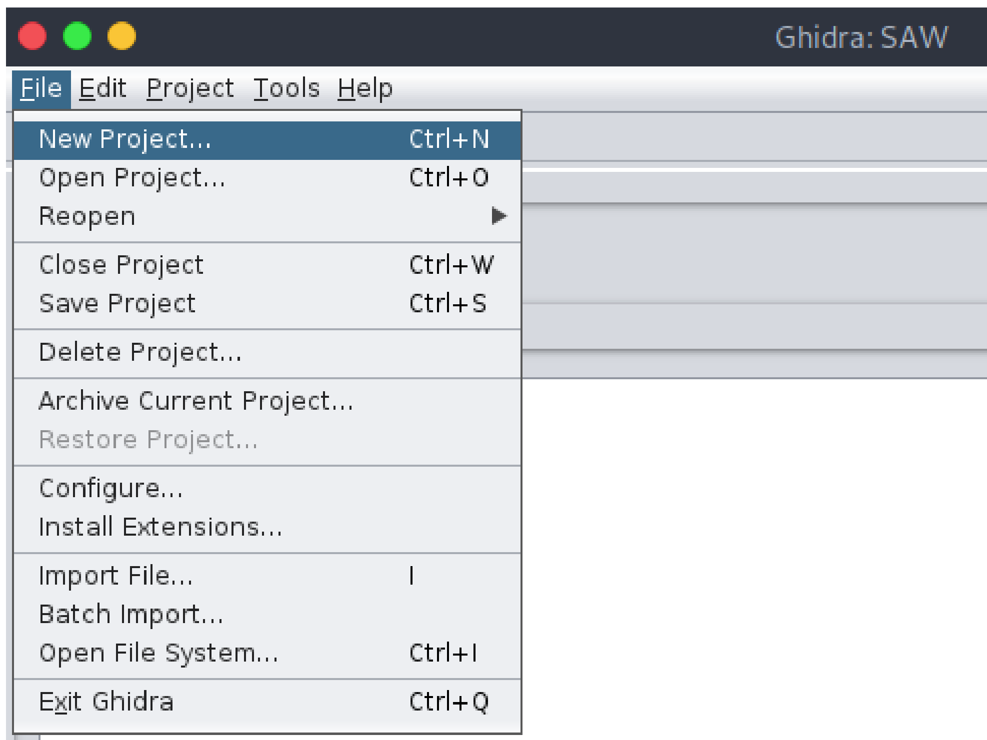


Figure 100 Reverse03 Ghidra New Project

On the pop-up window, we click next. Then we create the project directory and click finished.

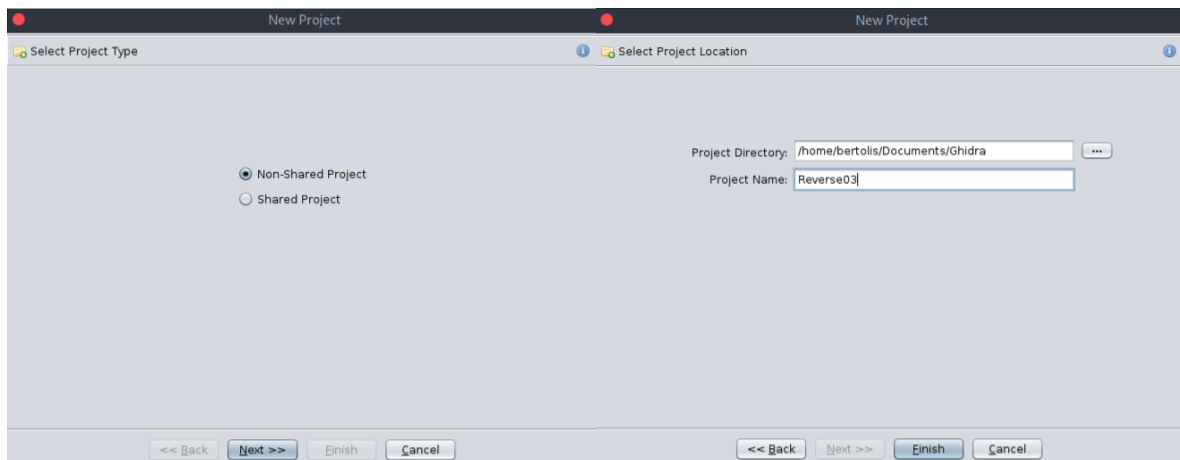


Figure 101 Reverse03 Ghidra Project Properties

Finally, we select the project we just created and click on the small green dragon icon to enter the **CodeBrowser** mode.

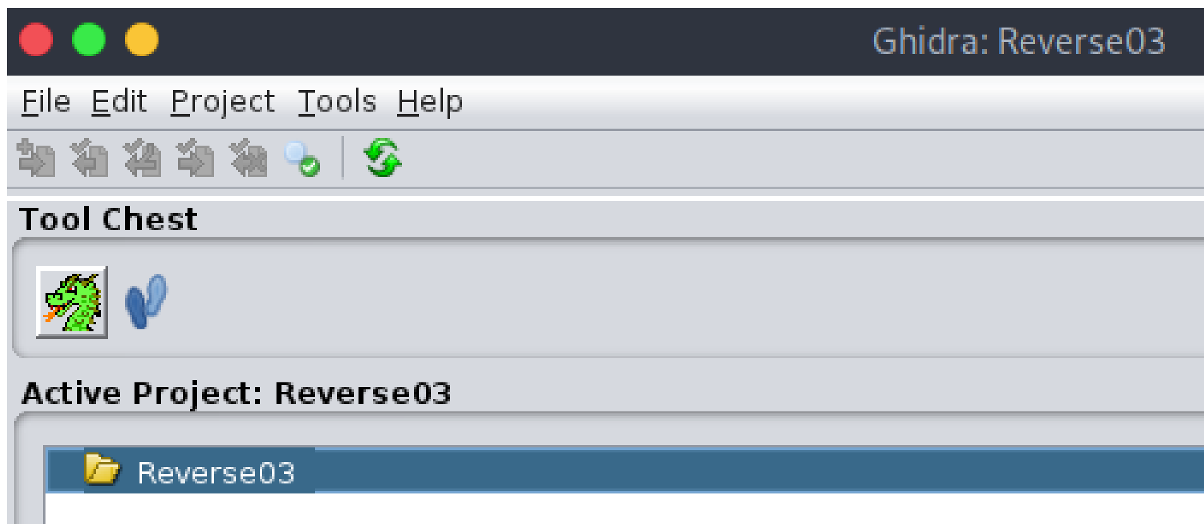


Figure 102 Reverse03 Start Project

Once we have entered the **CodeBrowser** mode, press “i” to import new file. Then, we navigate to the path where the **libreverse03.so** file is located, we select it and click the “Select File To Import” button.

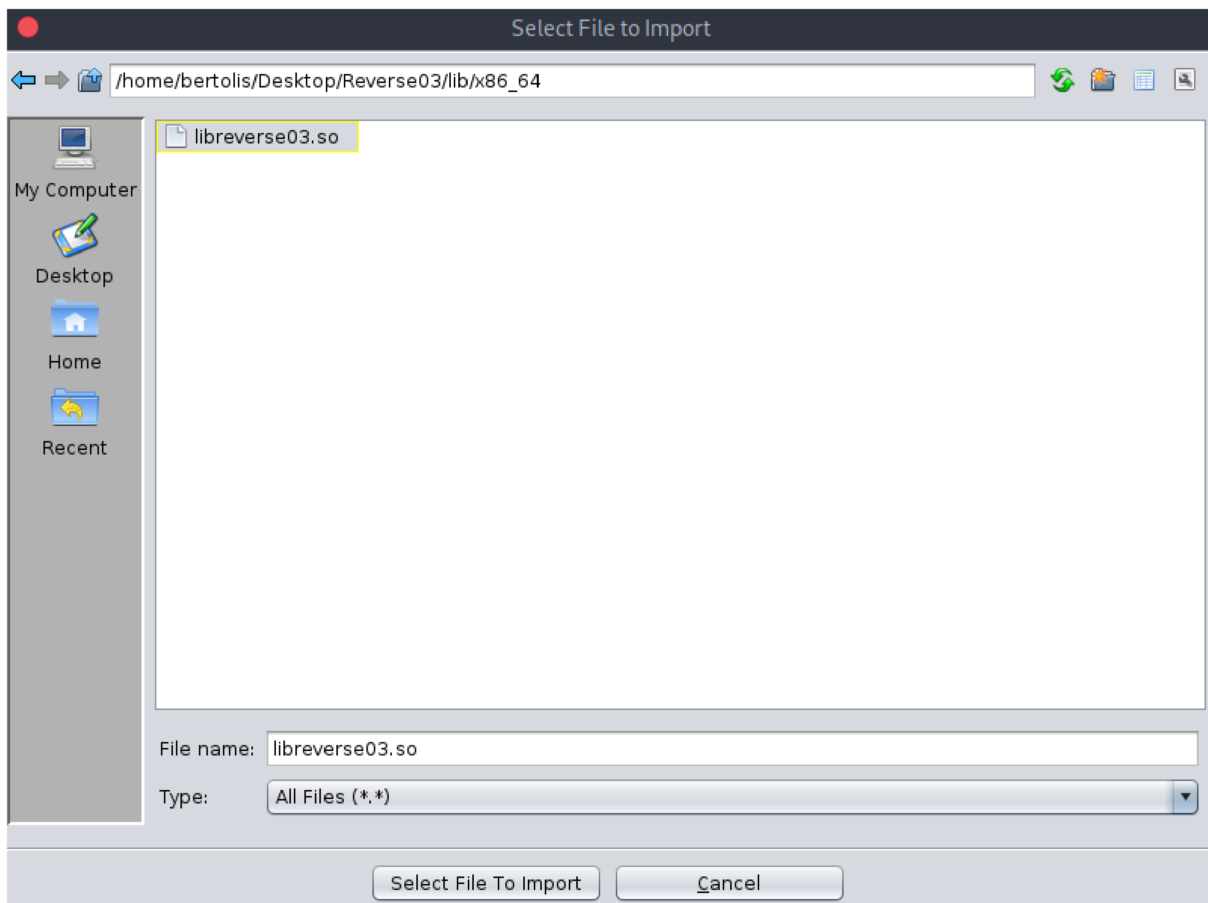


Figure 103 Reverse03 Load File

Next, we click **OK** on the next window, leaving the selected values.

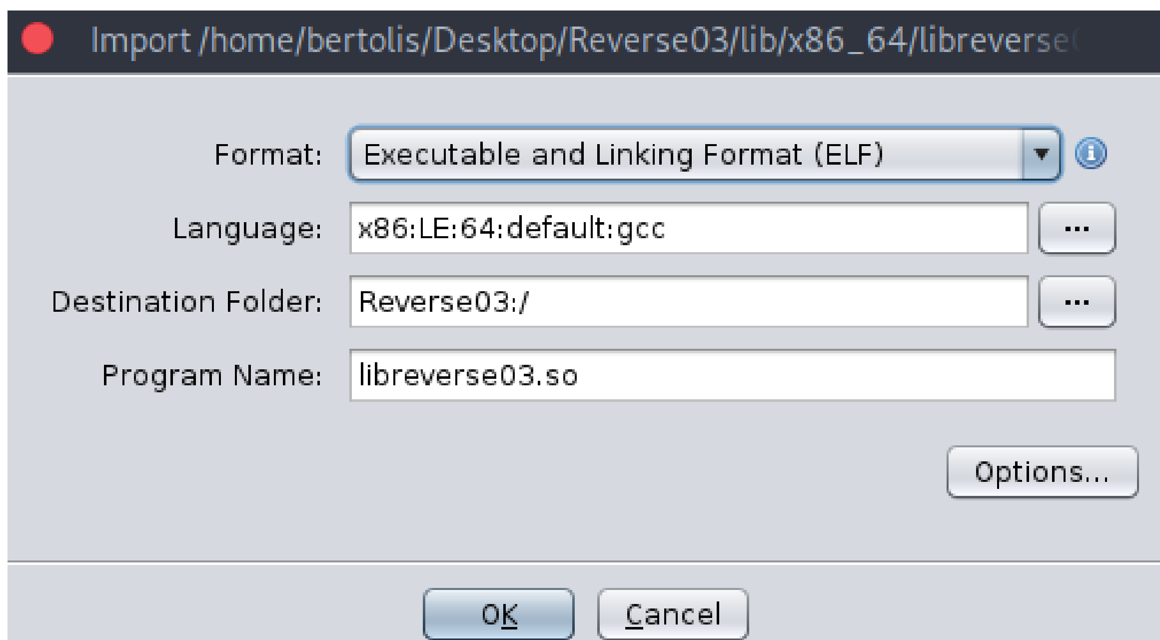


Figure 104 Reverse03 Ghidara File Properties

On the next pop-up window, we click **Analyze**.

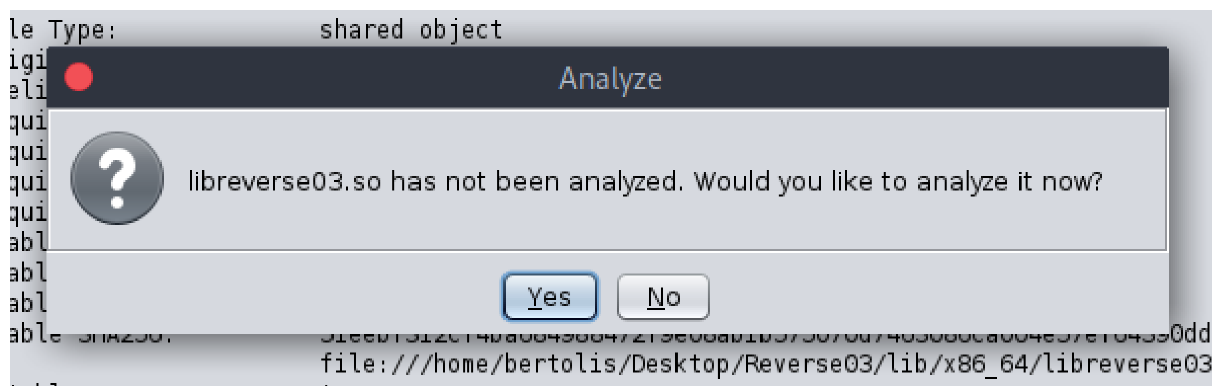


Figure 105 Reverse03 Ghidra Analyze

Once again, on the **Analyze Options** window, we click on the **Analyze** button.

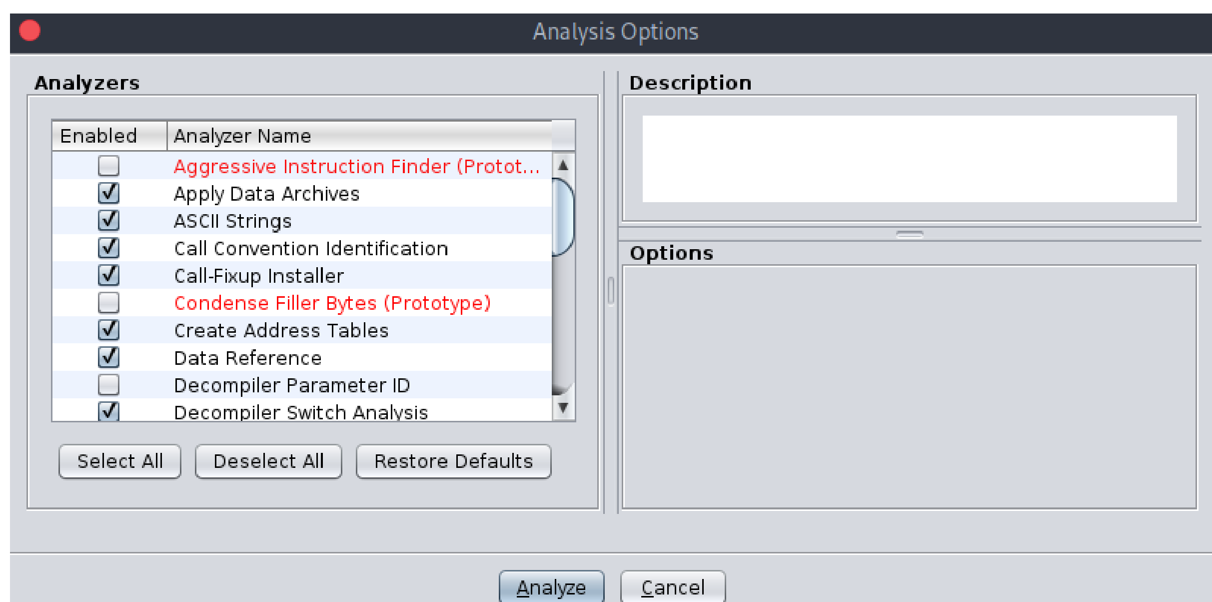


Figure 106 Reverse03 Ghidra Analazy Options

Finally, click **OK** on the **Summary** window.

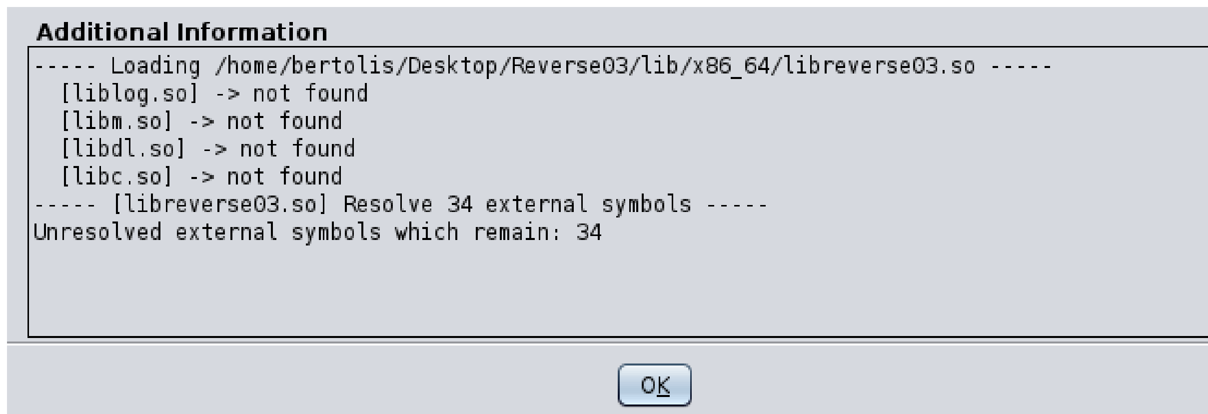


Figure 107 Reverse03 Ghidra Summary

Once we are done with the import process, the **CodeBrowser** window should look like this.

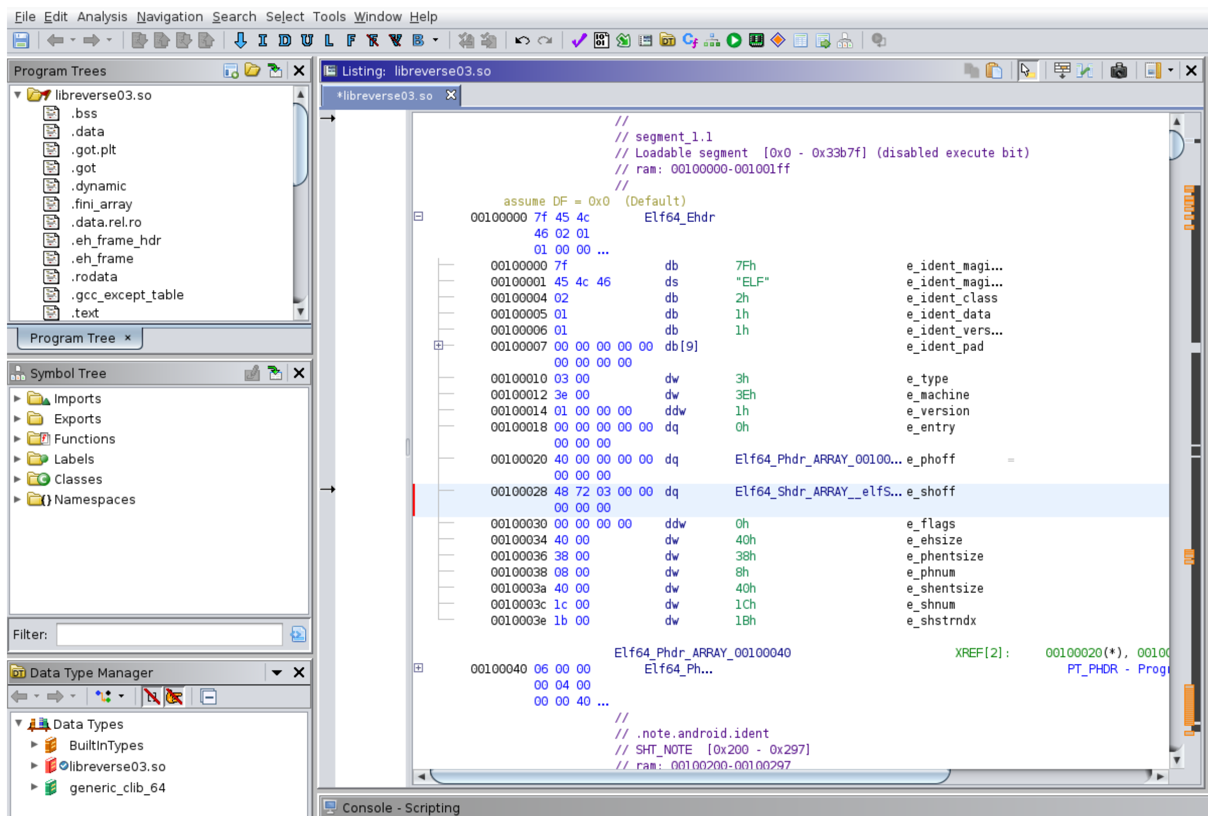


Figure 108 Reverse03 CodeBrowser

On the left side menu, under the **Functions** tab, we can see the function **java.com.example.reverse03_MainActivity_stringFromJNI**. Double click on it and navigate on the right section of the window. This section contains the pseudo code of this function. And we can try to read it.

Decompile: Java_com_example_reverse03_MainActivity_stringFromJNI - (libreverse03.so)

```
286   basic_string.conflict local_28 [16];
287   void *local_18;
288   long local_10;
289
290   local_10 = *(long *)(in_FS_OFFSET + 0x28);
291   basic_string<decltype(nullptr)>
292             ((basic_string<char,std--__ndk1--char_traits<char>,std--__ndk1--allocator<char>> *)
293              local_28,"U");
294   basic_string<decltype(nullptr)>
295             ((basic_string<char,std--__ndk1--char_traits<char>,std--__ndk1--allocator<char>> *)
296              local_40,"N");
297   basic_string<decltype(nullptr)>(&local_58,"I");
298   basic_string<decltype(nullptr)>(&local_70,"P");
299   basic_string<decltype(nullptr)>(&local_88,"I");
300   basic_string<decltype(nullptr)>(&local_a0,"{");
301   basic_string<decltype(nullptr)>(&local_b8,"n");
302   basic_string<decltype(nullptr)>(&local_d0,"4");
303   basic_string<decltype(nullptr)>(&local_e8,"t");
304   basic_string<decltype(nullptr)>(&local_100,"1");
305   basic_string<decltype(nullptr)>(&local_118,"v");
306   basic_string<decltype(nullptr)>(&local_130,"3");
307   basic_string<decltype(nullptr)>(&local_148,"_");
308   basic_string<decltype(nullptr)>(&local_160,"c");
309   basic_string<decltype(nullptr)>(&local_178,"0");
310   basic_string<decltype(nullptr)>(&local_190,"d");
311   basic_string<decltype(nullptr)>(&local_1a8,"3");
312   basic_string<decltype(nullptr)>(&local_1c0,"_");
313   basic_string<decltype(nullptr)>(&local_1d8,"1");
314   basic_string<decltype(nullptr)>(&local_1f0,"s");
315   basic_string<decltype(nullptr)>(&local_208,"_");
316   basic_string<decltype(nullptr)>(&local_220,"n");
317   basic_string<decltype(nullptr)>(&local_238,"0");
318   basic_string<decltype(nullptr)>(&local_250,"t");
319   basic_string<decltype(nullptr)>(&local_268,"_");
320   basic_string<decltype(nullptr)>(&local_280,"3");
321   basic_string<decltype(nullptr)>(&local_298,"n");
322   basic_string<decltype(nullptr)>(&local_2b0,"0");
323   basic_string<decltype(nullptr)>(&local_2c8,"u");
324   basic_string<decltype(nullptr)>(&local_2e0,"g");
325   basic_string<decltype(nullptr)>(&local_2f8,"h");
326   basic_string<decltype(nullptr)>(&local_310,"}");
327   operator+<char,std--__ndk1--char_traits<char>,std--__ndk1--allocator<char>>
328             ((__ndk1 *)local_6e0,local_28,local_40);
```

Figure 109 Reverse03 Flag

After scrolling a little bit down reveals the characters of the potential password that is returned. Let's check if this is the string that is returned from this function, and thus, the login password. To do so, we start the app on the device, then type the password that we just found and press **LOGIN**.

Figure 110 Reverse03 Login Screen

This is successful. We have found the hardcoded password.

Flag: UNIPI{n4t1v3_c0d3_1s_n0t_3n0ugh}

Mitigations

Hiding hard coded strings inside native code, is not the best security practice as we can see. Similarly to the Reverse02 challenge, using encrypted or remote authentication and enforcing secure communication, prevents issues like the one described above.

Objective

The objective of this scenario is to learn how to perform dynamic code instrumentation using the **Frida** framework.

Description

A colleague of mine forgot his master password of his password manager app, and lost access to valuable data. I already performed static analysis by reversing the APK file in order to get the password, but it seems that it is well hidden. If only there was another way to get the password.

Difficulty: Hard

Flag: UNIPI{n0th1ng_3sc4p3s_fr1d4}

Release: 8a100979c0e87f814d84d702caccac899853e99ec15f576be33a57b5d88ed1ca

Challenge

Start the Android Emulator that we have already set up in the previous chapter, and make sure that the emulator is attached to the ADB, by executing the following command.

```
adb devices
```

```
adb devices
List of devices attached
emulator-5554   device
```

Figure 111 Reverse04 ADB List Devices

Once we have confirmed the above step, unzip the **Reverse04.apk.zip** file and install the extracted file **Reverse04.apk** on the device, by issuing the following commands.

```
unzip Reverse04.apk.zip
adb install Reverse04.apk
```

```
unzip Reverse04.apk.zip
Archive:  Reverse04.apk.zip
  inflating: Reverse03.apk

adb install Reverse04.apk
Performing Streamed Install
Success
```

Figure 112 Reverse04 Unzip APK

Then, we can find the installed application in the device's menu and tap on to start it.

Password Manager

Please enter your master password to login to the app.

Master password

LOGIN

Figure 113 Reverse04 App

This is a password manager application. As the description implies, we are tasked to restore the password that is stored in the app's source code. We also know that static analysis by reversing the APK file has been performed. Let's try to examine the application by performing dynamic analysis techniques. But first, we will need to examine the APK statically and gather some information about the app.

Using the JADX tool, as we have already described in the Reverse01 challenge, we can read the some of the source code of the app. First, we convert the APK into a JAR file.

```
d2j-dex2jar Reverse04.apk
```

```
d2j-dex2jar Reverse04.apk
dex2jar Reverse04.apk -> ./Reverse04-dex2jar.jar
```

Figure 114 Reverse04 DEX2JAR

Then, we open it with JADX, we load the JAR file, and we navigate to the **MainActivity**.

```
jadx-gui
```

```
    @Override // android.view.View.OnClickListener
    public void onClick(View view) {
        if (MainActivity.this.f2389o.getText().toString().equals(MainActivity.this.stringFromJNI())) {
            MainActivity.this.startActivity(new Intent(MainActivity.this, ManageActivity.class));
            return;
        }
        Toast.makeText(MainActivity.this, "Wrong password!", 1).show();
    }
}
```

Figure 115 Reverse04 JADX

We can see, there is an **if()** statement that compares the user's input with the value that is returned from the function **stringFromJNI()**. The message "Wrong password!" is also displayed, ensuring that this is a password validation statement.

As we saw in the Reverse03 challenge while searching online the official Android documentation, this is a function that is used when we create C/C++ application using the NDK (Native Development Kit). These C/C++ files can be shared libraries that are used from the app. When the system builds the library, it adds the prefix **lib** and the extension **.so**.

Further enumeration with Apktool, as we saw in the Reverse02 challenge, we can extract some files from the APK.

```
wget https://bitbucket.org/iBotPeaches/apktool/downloads/apktool_2.6.0.jar
```

```
Resolving bbuseruploads.s3.amazonaws.com
(bbuseruploads.s3.amazonaws.com)... 54.231.129.145
Connecting to bbuseruploads.s3.amazonaws.com
(bbuseruploads.s3.amazonaws.com)|54.231.129.145|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 19964258 (19M) [application/x-java-archive]
Saving to: 'apktool_2.6.0.jar'

apktool_2.6.0.jar                100%
[=========================================================>]  19,04M
1,42MB/s    in 30s
```

Figure 116 Reverse04 Download APKTool

Once it's downloaded, we run the following command to extract the files.

```
java -jar apktool_2.5.0.jar d Reverse04.apk
```

```
java -jar apktool_2.6.0.jar d Reverse04.apk
I: Using Apktool 2.6.0 on Reverse04.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file:
/home/bertolis/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

Figure 117 Reverse04 APKTool Decompile

Enumerating the extracted files reveals the shared library file **Reverse04/lib/x86_64/libreverse04.lib**. As we saw earlier in the Reverse03 challenge, **.so** files are shared libraries used by programs installed on Linux and Android, and are written in

C/C++. Since **stringFromJNI()** is a function used from shared library files (.so), and the user's input is compared with the value that is returned from the **stringFromJNI()** function, it is very likely for the shared library **libreverse04.so** to be used by this app to fetch the password.

Before we start the dynamic analysis, there is one more thing we need not find. That is the full name of the function **stringFromJNI()** that is declared in the C++ file. Since we have already decompiled the APK file, we can find this information inside the **Reverse04/lib/x86_64/libreverse04.so** file. Although this file is still compiled, the strings are readable. We open the file using **vim** and search for "stringFromJNI".

```
vim Reverse04/lib/x86_64/libreverse04.so
```

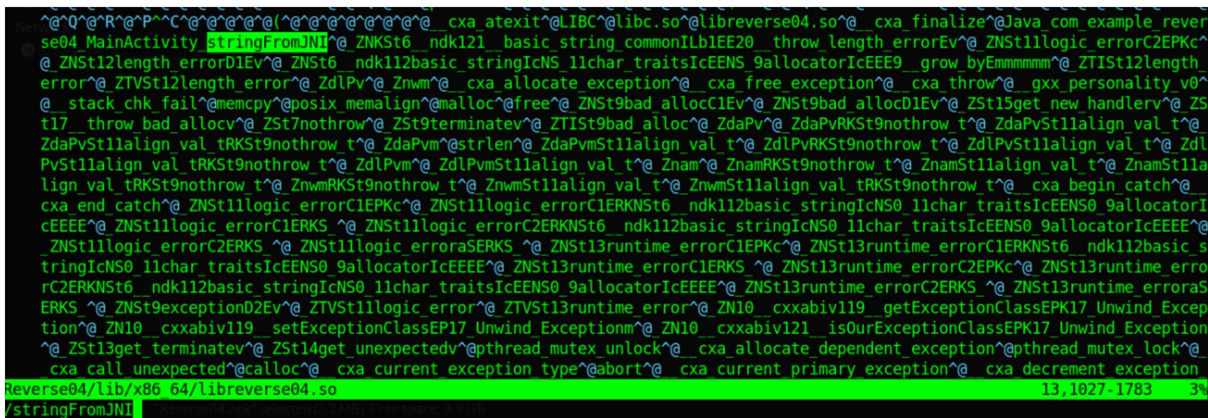


Figure 118 Reverse04 Shared Object Content

As we can see, the full name of the function is **Java_com_example_reverse04_MainActivity_stringFromJNI**.

Now that have some information about the app, let's try to hook the value returned by the function at run time. For this, we are going to use [Frida](#).

Frida is a dynamic code instrumentation toolkit. It lets you inject snippets of JavaScript or your own library into native apps, at runtime [48]. Essentially, apps can run through Frida by running the Frida [server](#) on the device. Then then we can install Frida tools on our host and

execute commands and scripts. Note that the version of Frida server should be the same version with the Frida tools.

Running Frida server on the device can be achieved as following. First, we download the server for android locally.

```
wget https://github.com/frida/frida/releases/download/15.1.1/frida-server-15.1.1-android-x86_64.xz
```

Figure 119 Reverse04 Download Frida Server

Once it’s downloaded, we can use the following command to decompress it.

```
xz -d -v frida-server-15.1.1-android-x86_64.xz
```

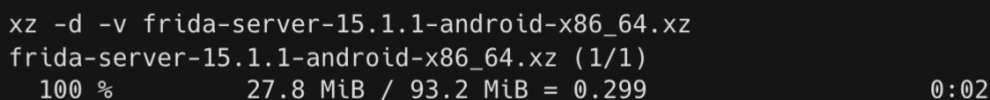


Figure 120 Reverse04 Decompress Frida Server

Then, we must push the extracted file into the Android device. Let's use the world writable directory **/data/local/tmp/** and save it as **frida-server**.

```
adb push frida/frida-server-15.1.1-android-x86_64 /data/local/tmp/frida-server
```

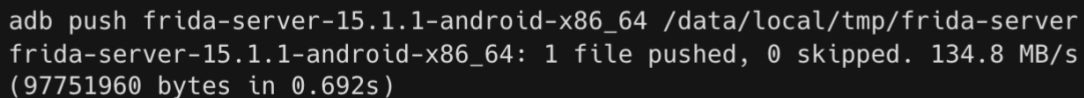


Figure 121 Reverse04 ADB push frida server

Frida server is successfully transferred on the device. Now we need to install Frida tools. On our host machine, we execute the following command.

```
pip install frida-tools
frida --version
```

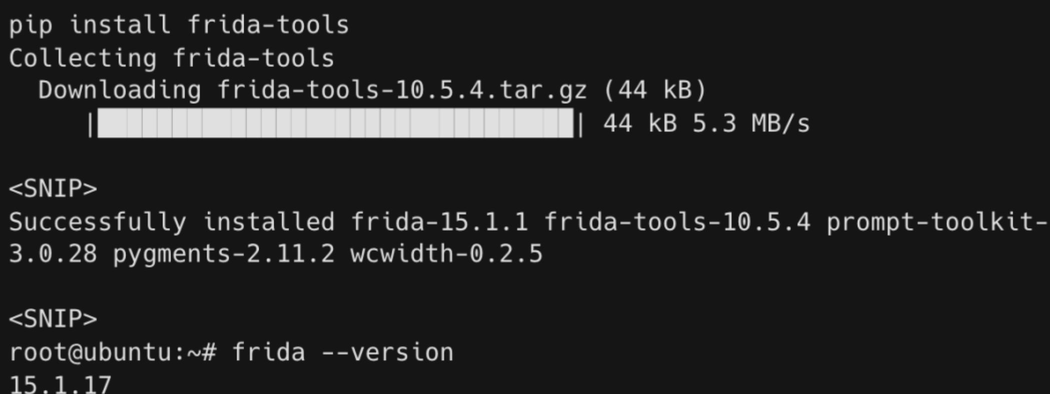


Figure 122 Reverse04 Install Frida Tools

Now that both Frida server and Frida tools are installed, we can go on and search online for scripts that allows function hooking. Since we assume that the password might be the value that is returned from the **stringFromJNI()** function, let's form our search like this.

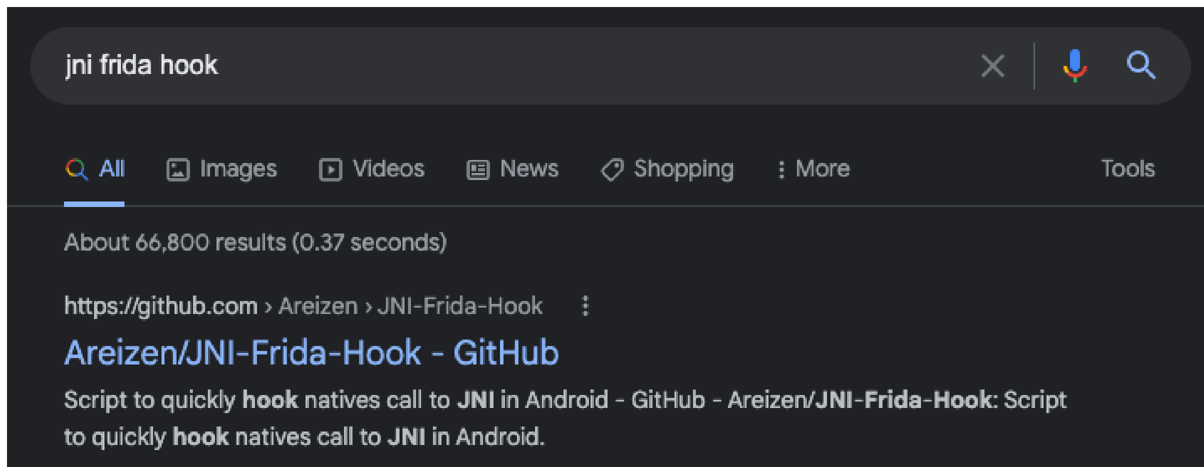


Figure 123 Reverse04 JNI Frida Hook

Searching online for **jni frida hook** reveals this GitHub repository. As we said earlier, using Frida tools we can execute scripts written in Javascript, while the application is running. This repository provides a script that allows us to have an overview of JNI (Java Native Interface) called by a function, and hook them [49]. Let's clone this repository, by issuing the following command.

```
git clone https://github.com/Areizen/JNI-Frida-Hook.git
```

```
git clone https://github.com/Areizen/JNI-Frida-Hook.git
Cloning into 'JNI-Frida-Hook'...
remote: Enumerating objects: 30, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 30 (delta 2), reused 9 (delta 2), pack-reused 21
Receiving objects: 100% (30/30), 9.53 KiB | 3.18 MiB/s, done.
Resolving deltas: 100% (7/7), done.

test ls -l JNI-Frida-Hook
total 16
-rw-r--r--  1 bertolis  staff  1412 Feb 23 13:56 README.md
-rw-r--r--  1 bertolis  staff  3092 Feb 23 13:56 agent.js
drwxr-xr-x  3 bertolis  staff    96 Feb 23 13:56 utils
```

Figure 124 Reverse04 Git Clone Hooking Script

As the instructions indicate, we need to change the following.

```
Usage

Fill library name and function name in agent.js

library_name = "" // ex: libsqlite.so
function_name = "" // ex: JNI_OnLoad
```

Figure 125 Reverse04 Hooking Script Variables

In our case, the library name is **libreverse04.so**, and the function name is **Java_com_example_reverse04_MainActivity_stringFromJNI**. Once we have changed them, the **agent.js** script should look like this.

```
agent.js
1   const jni = require("./utils/jni_struct.js")
2
3   var library_name = "libreverse04.so" // ex: libsqlite.so
4   var function_name = "Java_com_example_reverse04_MainActivity_stringFromJNI" // ex: JNI_OnLoad
5   var library_loaded = 0
6
7
```

Figure 126 Reverse04 Script Library and Function Names

The value **FindClass** in the following snippet of code, will hook all the functions of the **JNI**.

```
Interceptor.attach(jni.getJNIFunctionAdress(jnienv_addr,"FindClass"),{
    onEnter: function(args){
        console.log("env->FindClass(\"" + Memory.readCString(args[1]) + "\")")
    }
})
```

Figure 127 Reverse04 Script Find Functions

Next, we save and compile the **agent.js** file, using the following command.

```
npm install frida-compile
```

```
npm install frida-compile
npm WARN deprecated querystring@0.2.0: The querystring API is considered
Legacy. new code should use the URLSearchParams API instead.

added 255 packages, and audited 256 packages in 6s
<SNIP>
```

Figure 128 Reverse04 Install Frida Compile

```
node node_modules/.bin/frida-compile agent.js -o _agent.js
```

The file **_agent.**js is now created. Before we run the script, we need to find the package name of the app. From the emulated device, spot the app and tap on it to start. Then type the following on the terminal.

```
adb shell ps | grep reverse04
```

```
adb shell ps | grep reverse04
u0_a209      16360  1747 4923632 112956 SyS_epoll_wait      0 S
com.example.reverse04
```

Figure 129 Reverse04 Get Package Name

The package name of the app is **com.example.reverse04**. Now that we have everything we need to run the script, we can start the Frida server that we pushed earlier into the devices.

```
adb shell chmod 755 /data/local/tmp/frida-server
adb shell /data/local/tmp/frida-server &
```

```
adb shell /data/local/tmp/frida-server &
[1] 60481
```

Figure 130 Reverse04 Start Frida Server

Finally, we execute the script to hook the JNI functions. The following command will start the app from within the Frida server that we pushed into the device earlier, and then try to hook the JNI function at run time. First, we close the app from the device. Then we issue the following command.

```
frida -U -f com.example.reverse04 -l _agent.js --no-paus
```

```
frida -U -f com.example.reverse04 -l _agent.js --no-paus
     ____
    / _  |   Frida 15.1.1 - A world-class dynamic instrumentation
toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
Spawned `com.example.reverse04`. Resuming main thread!
[Android Emulator 5554::com.example.reverse04]-> [...] Loading library :
/data/app/com.example.reverse04-8Jb7ikTbG0CJKRbt-
leoLg==/lib/x86_64/libreverse04.so
[+] Loaded
[...] Hooking : libreverse04.so ->
Java_com_example_reverse04_MainActivity_stringFromJNI at 0x7d1396ad4bb0
[Android Emulator 5554::com.example.reverse04]->
```

Figure 131 Reverse04 Frida Hook Native Functions

We notice that the app on the devices has started. Let’s tap the **LOGIN** button and check the results.

```
[Android Emulator 5554::com.example.reverse04]-> [+] Hooked
successfully, JNIEnv base adress :0x7d13eaddb6b0
[+] Entered : NewStringUTF
[-] Detaching all interceptors
```

Figure 132 Reverse04 Hooked Functions

The function **NewStringUTF** has been successfully hooked. Let’s try to hook the return value of this function. To do so, we have to change the **findClass** instruction, with the function name **NewStringUTF** inside the **agent.js** file. The **agent.js** file should look like this.

```
Interceptor.attach(jni.getJNIFunctionAdress(jnienv_addr,"NewStringUTF"),{
    onEnter: function(args){
        console.log("env->NewStringUTF(\"" + Memory.readCString(args[1]) + "\")")
    }
})
```

Figure 133 Reverse04 Script Function Name

Once that is done, we save and compile the file once again.

```
node node_modules/.bin/frida-compile agent.js -o _agent.js
```

Once the **agent.js** file is created, we close the app from the device and run the following command once again.

```
adb shell /data/local/tmp/frida-server &
frida -U -f com.example.reverse04 -l _agent.js --no-paus
```

Once the app starts, we click the **LOGIN** button. Looking at the results, we can see that the return value of the function **NewStringUTF** has been successfully hooked.

Figure 134 Reverse04 Flag

Flag: UNIPI{n0th1ng_3sc4p3s_fr1d4}

Mitigations

Hardcoded passwords should not be used as a practice when creating an app. Using encrypted or remote authentication and enforcing secure communication, as the official android documentation suggests, prevents issues like the one described above.

Traffic Analysis

In this type of challenges, the player should perform traffic analysis in order to find bad security practices. A flag, a specific predefined word of the format UNIPI{s0m3_r4nd0m_t3xt!}, will be indicating this bad practice or the results of it.

TrafficAnalysis01

Objective

The objective of this scenario is to learn how to set up a proxy server using Burp Suite on an Android emulator, and capture HTTP requests that the application sends to a remote server.

Description

My father started using a password manager. Unfortunately, he forgot his master password. Fortunately, he can still login to the app because the password is stored. I have figured out that the login authentication is done remotely. If only there was a way to capture the password.

Difficulty: Easy

Flag: UNIPI{pl@1nt3xt_1nt3rc3pt1on_0-1}

Release: cff6028033d716617f9ea87ae638025fd3105cae9f7c3feb41429050b5a80e59

Challenge

Start the Android Emulator that we have already set up in the previous chapter, and make sure that the emulator is attached to the ADB, by executing the following command.

```
adb devices
```

```
adb devices
List of devices attached
emulator-5554   device
```

Figure 135 TrafficAnalysis ADB List Devices

Once we have confirmed the above step, unzip the **TrafficAnalysis01.apk.zip** file and install the extracted file **TrafficAnalysis01.apk** on the device, by issuing the following commands.

```
unzip TrafficAnalysis01.apk.zip
adb install TrafficAnalysis01.apk
```

```
unzip TrafficAnalysis01.apk.zip
Archive:  TrafficAnalysis01.apk.zip
  inflating: TrafficAnalysis01.apk

adb install TrafficAnalysis01.apk
Performing Streamed Install
Success
```

Figure 136 TrafficAnalysis Unzip APK

Then, we can find the installed application in the device’s menu, and tap on to start it.

Password Manager

Please enter your master password to login to the app.

LOGIN

Figure 137 TrafficAnalysis App

This is a password manager application. As the description implies, we are tasked to restore the master password, that is saved in the login form. Given the category and the hint from the description, that authentication is done remotely, it would be wise to try to intercept the potential HTTP request using Burp Suite.

Burp Suite is an integrated platform/graphical tool for performing security testing of web applications [50], and it is pre-installed in Parrot Linux. Alternatively, we can download it directly from the official website.

Let's configure our Android emulator to use Burp's proxy server. The first thing we need to know is the IP address of the host. To find it, type the following.

```
ifconfig
```

```
ifconfig

<SNIP>
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.6  netmask 255.255.255.0  broadcast
192.168.125.255
        inet6 fd15:4ba5:5a2b:1008:dd1:b187:6263:b315  prefixlen 64
scopeid 0x0<global>
        inet6 fe80::8c95:b856:cce3:a0a7  prefixlen 64  scopeid
0x20<link>
        ether 00:0c:29:25:de:f3  txqueuelen 1000  (Ethernet)
        RX packets 86902  bytes 112948150 (107.7 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 21504  bytes 2143917 (2.0 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
<SNIP>
```

Figure 138 TrafficAnalysis Get Local IP

The IP address of the host is **192.168.1.8**. Now we can go on and start Burp, and navigate to the **Proxy Listeners** dialog box, under **Proxy -> Options** tab. Then, select the entry with the interface **127.0.0.1:8080**, and click on **Edit**.

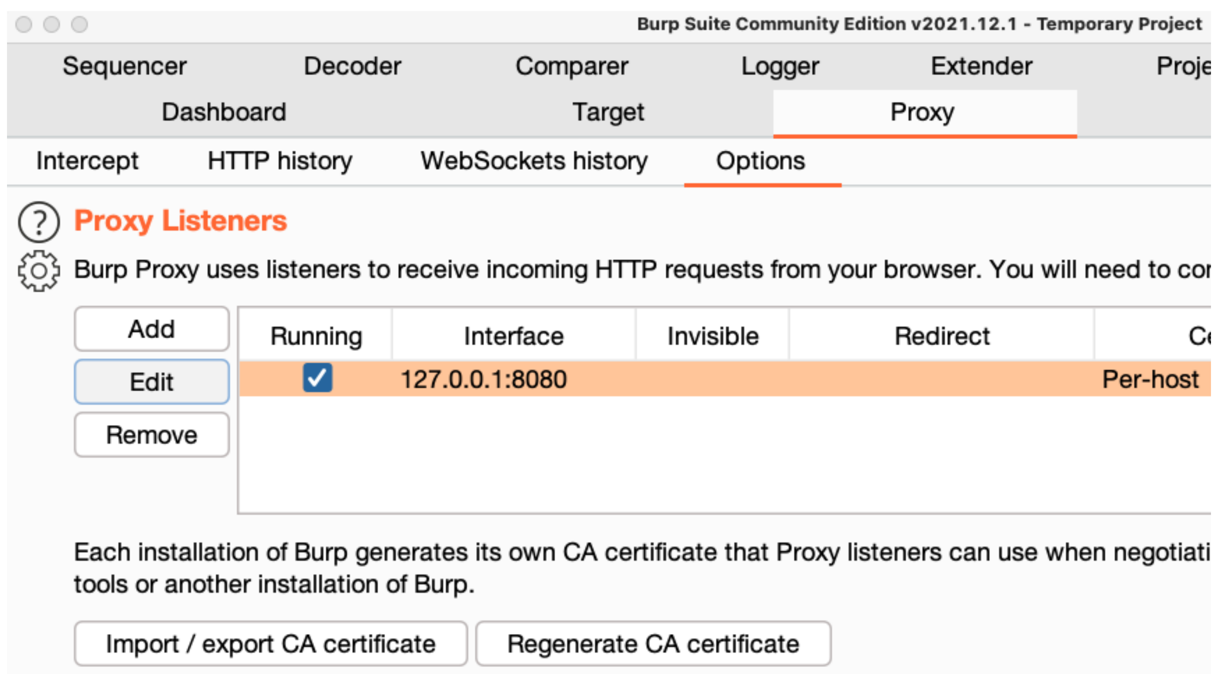


Figure 139 TrafficAnalysis Burp Proxy Tab

Once we are inside the **Edit** window, change the **Bind to port** option to **8090** instead of **8080**, and set the **Bind to address** to **Specific address**. Then from the drop down menu, select the IP address **192.168.1.8**.

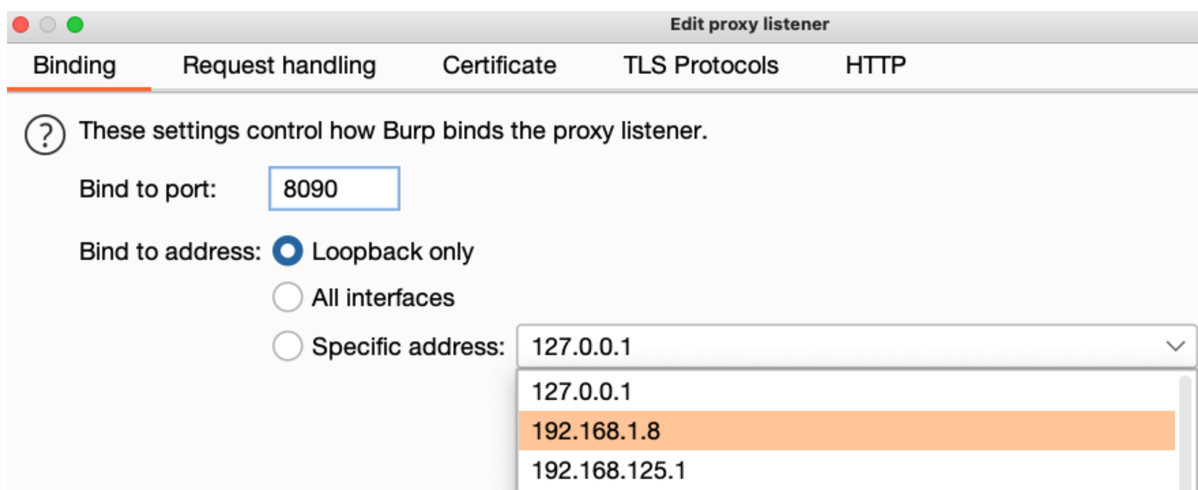


Figure 140 TrafficAnalysis Burp Binding IP

Finally, we click **OK**, and then we make sure that the **Intercept is on** button is toggled on, under the **Proxy -> Intercept** tab.

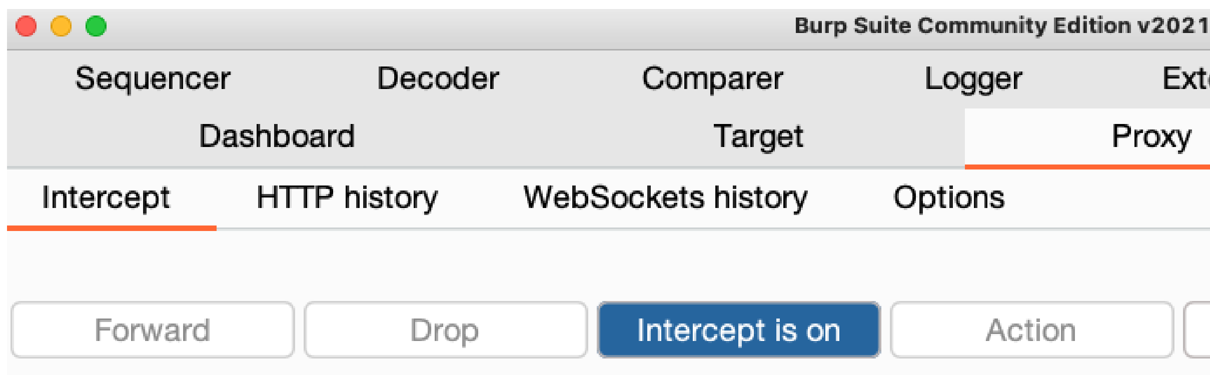


Figure 141 TrafficAnalysis Burp Intercept On

Once we are done, we can go on and configure the Android emulator to use the proxy server we just set up on Burp. First, we click on the three dots on the lower right of the device's vertical menu.

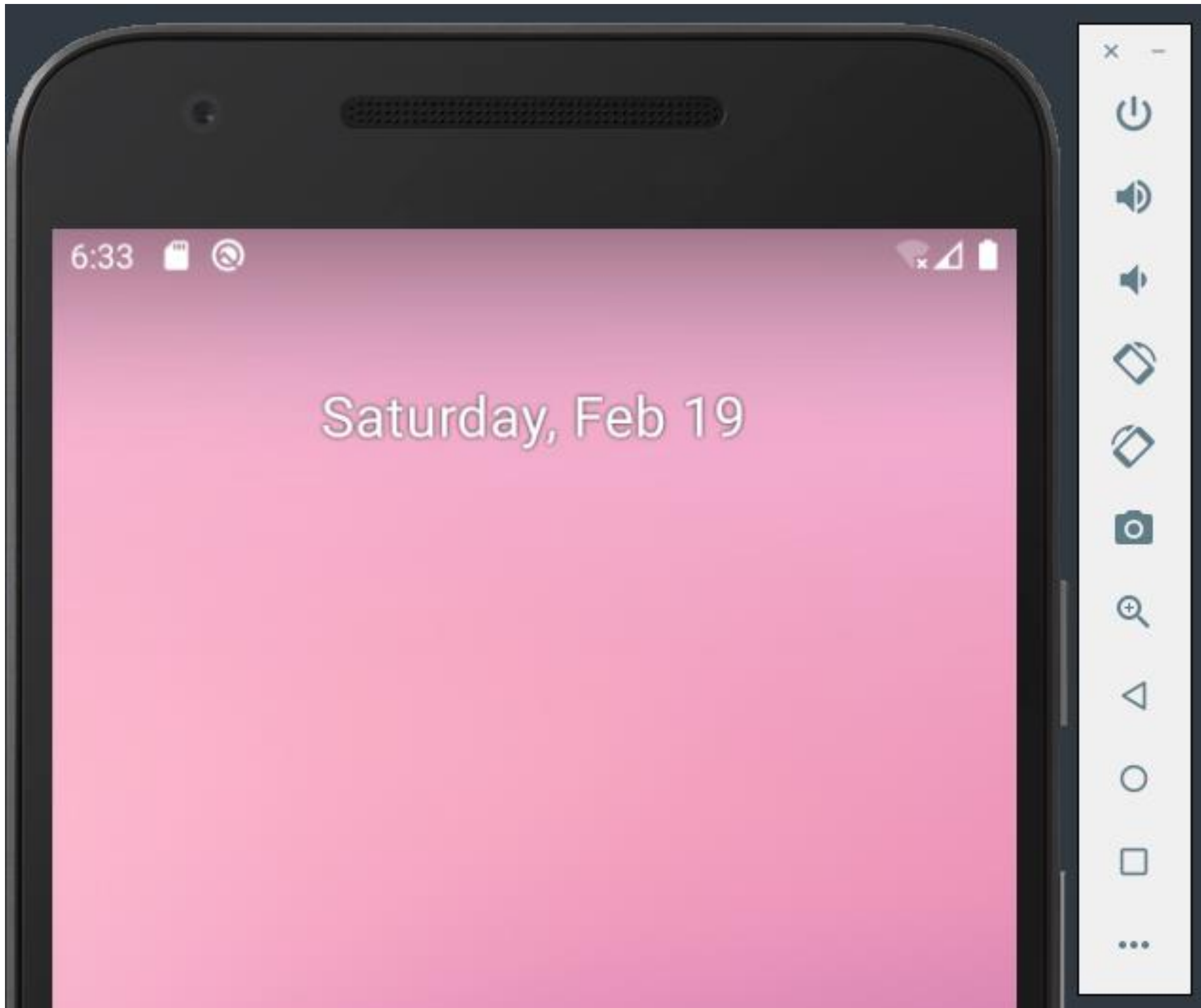


Figure 142 TrafficAnalysis Emulator Settings

Once that is clicked, the **Extended Controls** panel should pop up. From the vertical menu on the left of the window, select **Settings** and then click on the tab **Proxy**. Then uncheck the box that says **Use Android Studio HTTP proxy settings** and check the **Manual proxy configuration** box. Finally, set the values **192.168.1.8** and **8090** in the **Host name** and **Port number** fields accordingly, a click on the **Apply** button.

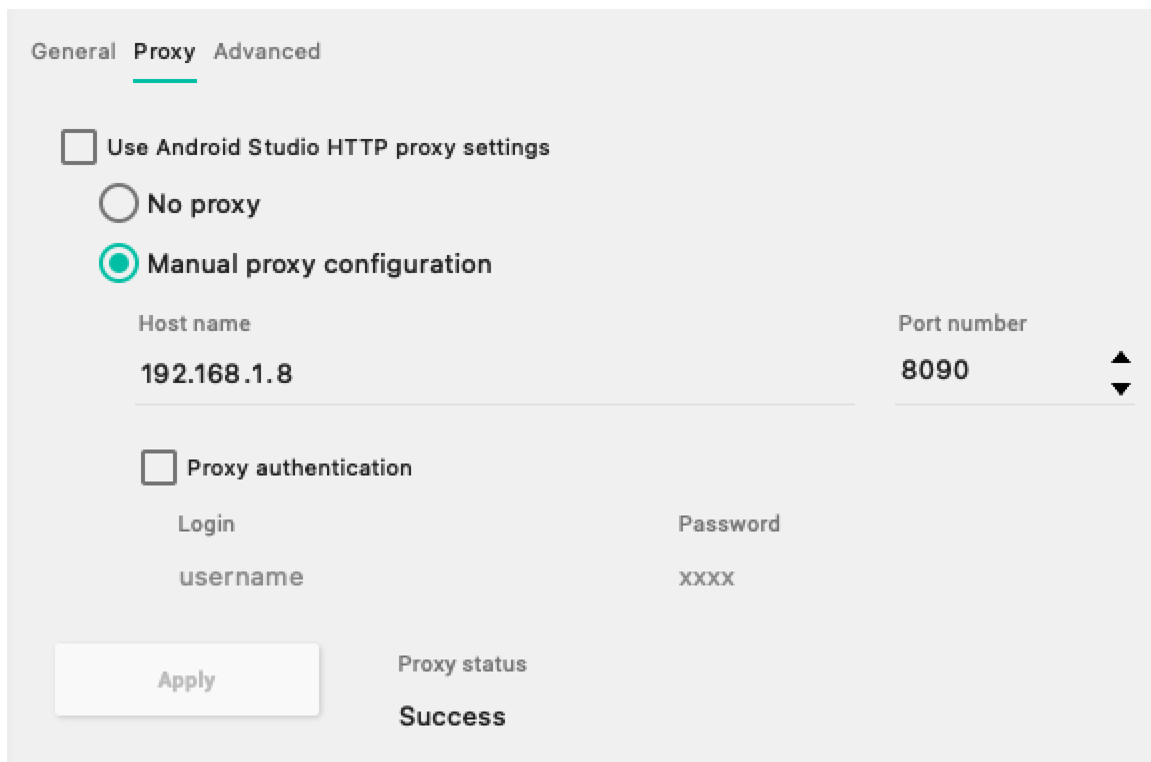


Figure 143 TrafficAnalysis Emulator Proxy Tab

If everything goes well, the **Proxy status** should set to **Success**. Now that everything is set up properly, start the application, make sure that the saved password is in the field, and click **LOGIN**.

Finally, if we look at the **Proxy -> Intercept** tab on Burp, we will see the intercepted POST request and the password in plain text, passed on the parameter **pass**.

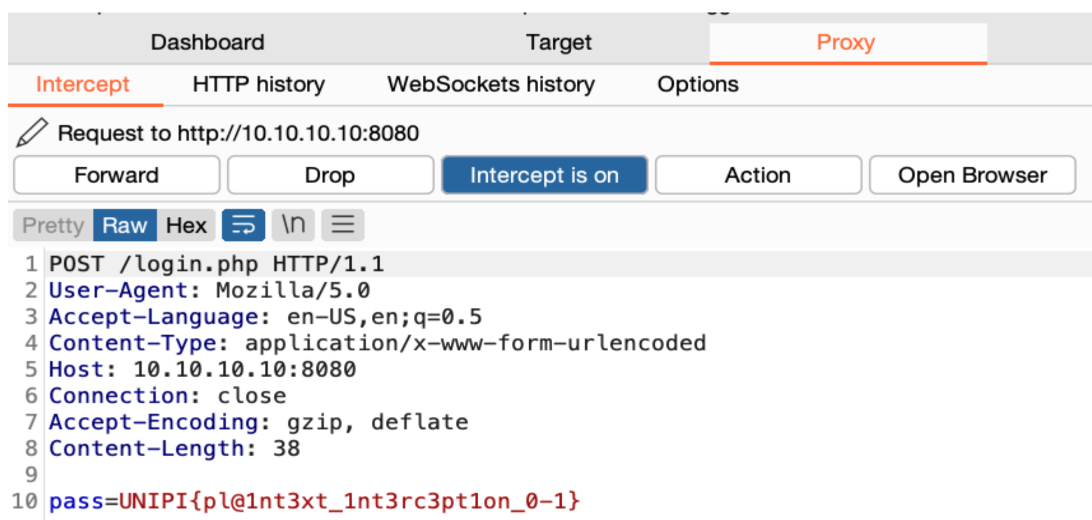


Figure 144 TrafficAnalysis Flag

Flag: UNIPI{pl@1nt3xt_1nt3rc3pt1on_0-1}

Mitigations

In order to mitigate issues like this, Secure Sockets Layer (SSL) can be used. Implementing SSL connection in Android applications can be done in various ways, as we can see in the official Android documentation [51].

Forensics

In this type of challenges, the player should load the disk image into forensics tool, and perform forensics analysis in order to find the data that are specified in the challenge's description. A flag, a specific predefined word of the format UNIPI{s0m3_r4nd0m_t3xt!}, will be indicating that these data have been found.

Forensics01

Objective

The objective of this scenario is to learn how to analyse an Android disk image, and extract text messages and data from databases, using Autopsy.

Description

We have managed to acquire the disk image of the suspect's Android phone. We have the suspicion that classified information is stored inside an application. Can you help us get this information?

Difficulty: Easy

Flag: UNIPI{1_th0ught_1t_w4s_s4f3}

Release: 473ed1d9cc629e1a4567a38e2114efae45e970551a58ccd887486a6e9a01e4cc

Challenge

As the description implies, we are tasked to extract classified information that was stored inside an application, installed in the suspect's phone.

In order to analyse the disk image, we have to download Autopsy. Autopsy is an open-source digital forensics platform [52]. Once we have downloaded and installed Autopsy in our Windows machine, we open it and click on **New Case**.

Figure 145 Forensics01 Start Autopsy

On the next two windows, we fill in the case name and the case number accordingly.

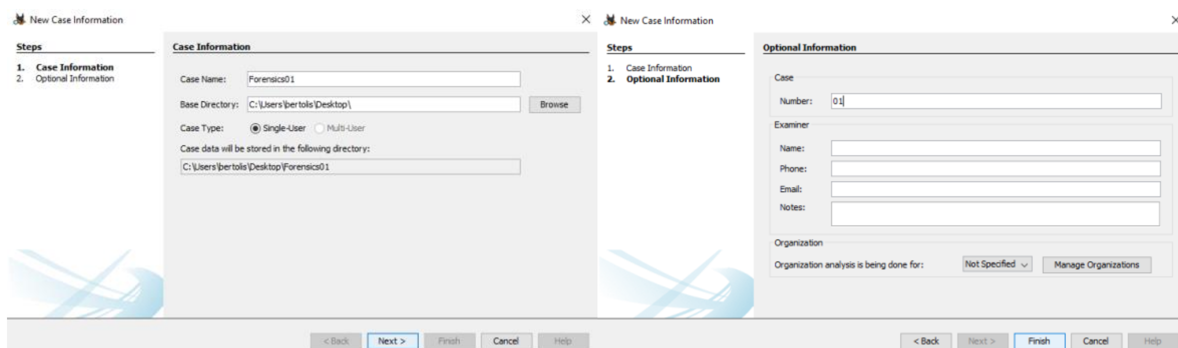


Figure 146 Forensics01 Case Name and Number

On the **Select Host** and **Select Data Source Type** windows, we leave the default values and click **Next**.

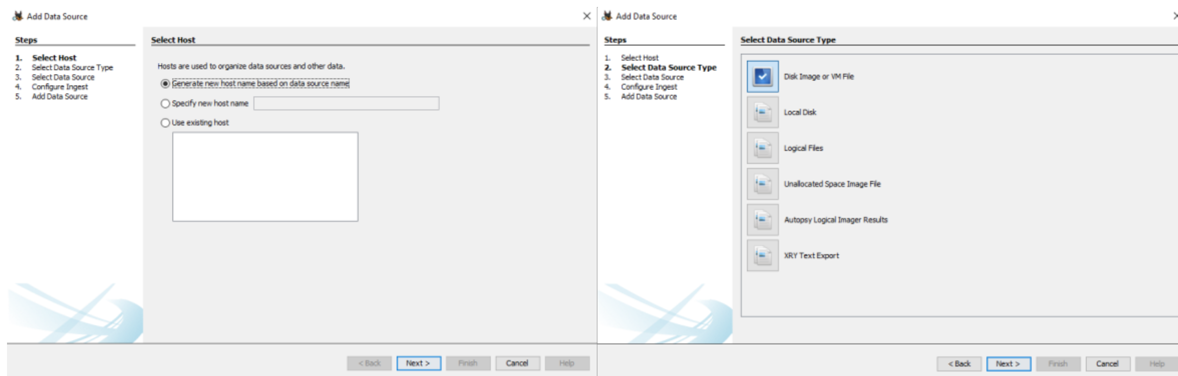


Figure 147 Forensics01 Host and Source Type

Then, on the **Select Data Source** window we click the **Browse button** on the right, we select the **Forensics01.dd** file, and we click **Next**.

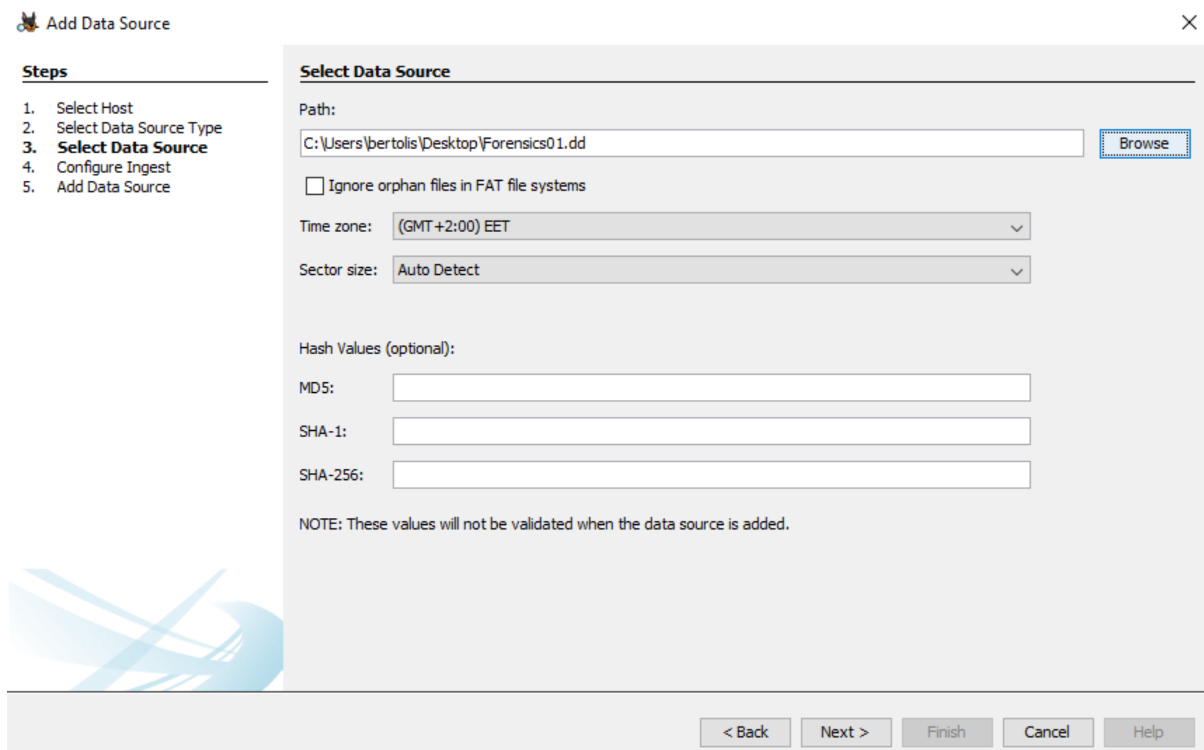


Figure 148 Forensics01 Import File

Finally, we click **Next** once again on the **Configure Ingest** window, and then **Finished**.

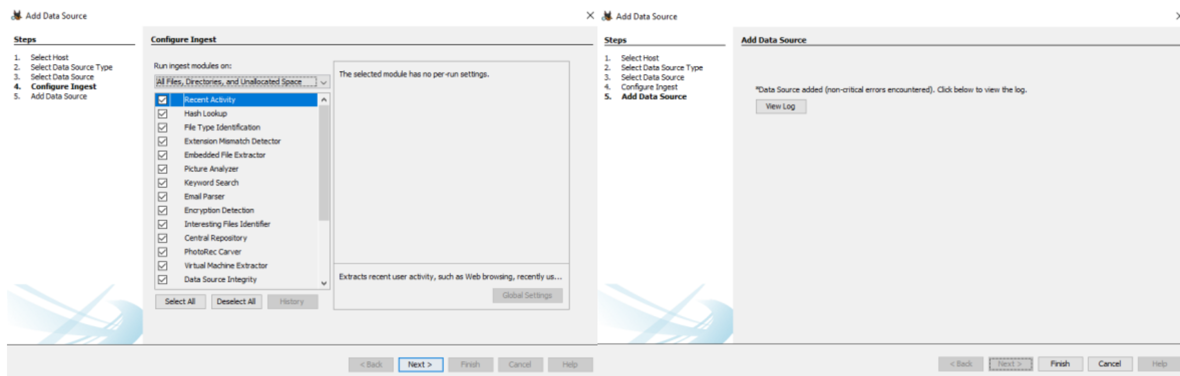


Figure 149 Forensics01 Configute Ingest

Once the loading bar on the bottom right is fully loaded, the window should be looking like this.

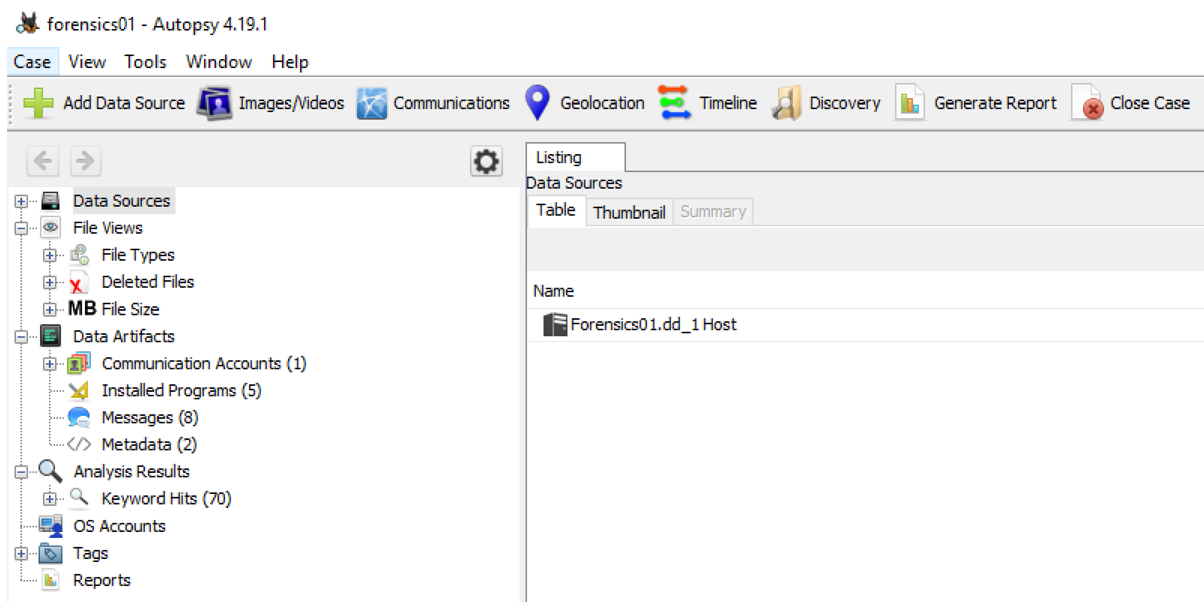


Figure 150 Forensics01 Autopsy Home Screen

Under **Communications -> Browse** tab, we can see the phone number **6505551212**. By selecting this number, we can then navigate to the **messages** tab on the right section of the window, and read the messages. Reading the conversation, we concluded that the one person keeps his credentials inside a password manager app.

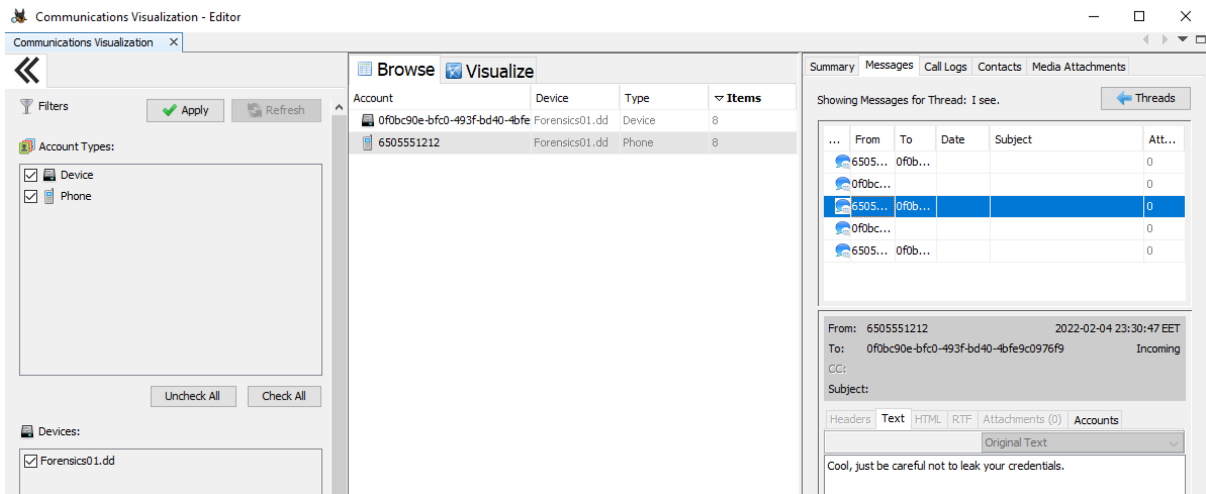


Figure 151 Forensics01 Communications Window

Let's search the local storage for installed password manager applications. On the folding menu on the left, under the **Data Sources -> Forensics01.dd_1 Host -> Forensics01.dd -> data -> com.example.passwordmanager -> databasesForensics01.dd**, we can see a file called **appDatabase**.

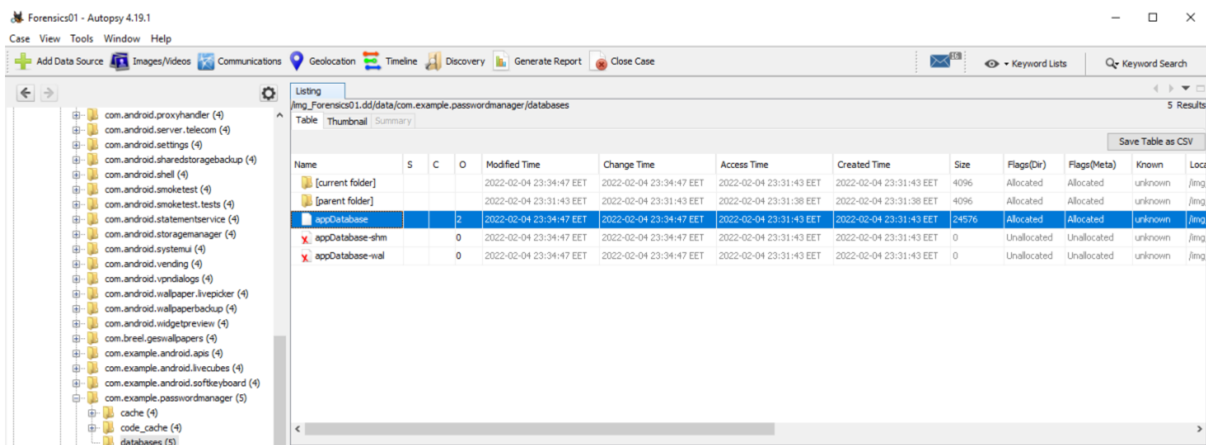


Figure 152 Forensics01 App Database

To save the file to our host machine, we right click on it, and then we click **Extract File(s)**.

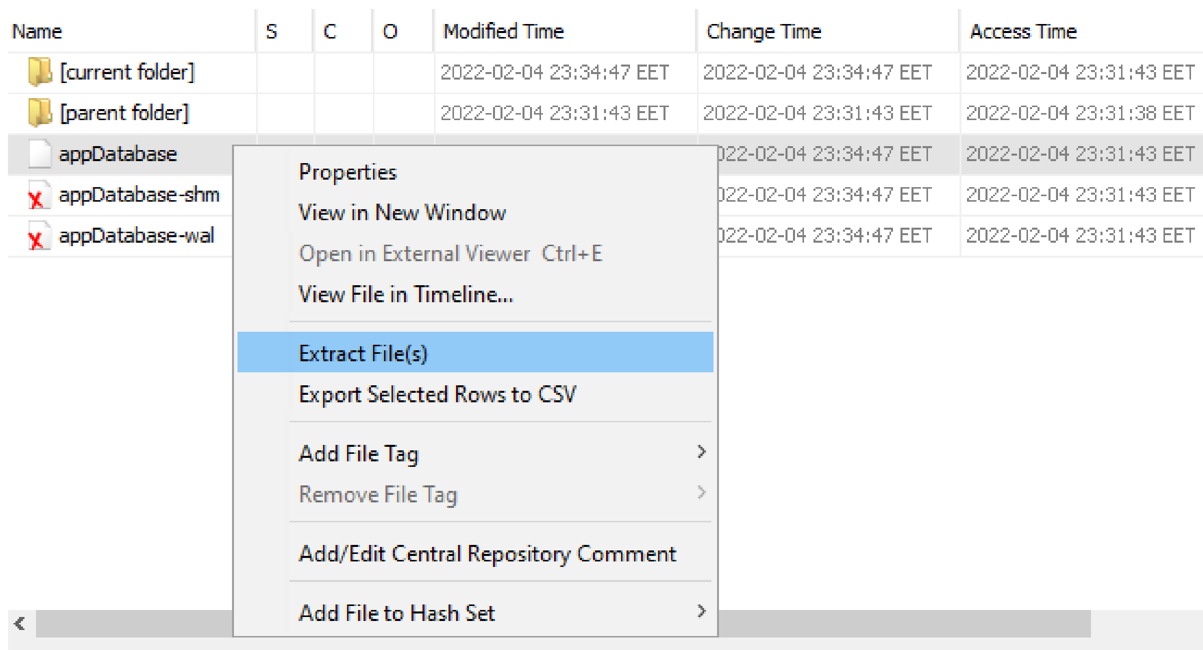


Figure 153 Forensics01 Extract Database

We save the extracted file in the Desktop folder, and then transfer it to our local Parrot Linux OS, in order to read it using an **sqlite3** client. Once we have it locally, we issue the following command to get into the sqlite interactive mode.

```
sqlite3 appDatabase
```

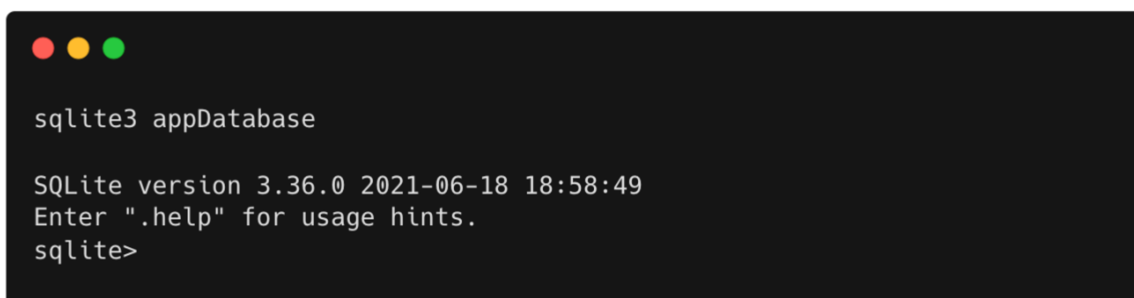


Figure 154 Forensics01 Open Database

Let’s now list the tables of the database, by executing the following command.

```
sqlite> .tables
```

```
sqlite> .tables
SiteCredentials    android_metadata   room_master_table
```

Figure 155 Forensics01 Get Tables

Finally, lets run the following query to get the data from the columns of this table.

```
sqlite> select * from SiteCredentials;
```

```
sqlite> select * from SiteCredentials;
https://secret-site.com|john42|UNIPI{1_th0ught_1t_w4s_s4f3}
```

Figure 156 Forensics01 Flag

The credentials for the website **https://secret-site.com** are revealed.

Flag: UNIPI{1_th0ught_1t_w4s_s4f3}

Forensics02

Objective

The objective of this scenario is to learn how to analyse an Android disk image, in order to read the web browsing data and restore deleted files, using Autopsy.

Description

We believe that suspicious files were downloaded from the deep web on the suspects mobile phone. After successfully acquiring the Android disk image from his phone, we searched for these files but they might have been already deleted. Can you help us restore them?

Difficulty: Easy

Flag: UNIPI{1_th0ught_1t_w4s_d3l3t3d!!!}

Release: da00c728ef40190883bb0a99cafdad37bf7b53b9e855c1b8cc9f2dd6f06cb20d

Challenge

As the description implies, we are tasked to restore potentially deleted files from the suspect’s Android phone.

In order to analyse the disk image, we have to download Autopsy, as we did on the previous challenge. Once everything is ready, we open it and click on **New Case**.

Figure 157 Forensics02 Start Autopsy

On the next two windows, we fill in the case name and the case number accordingly. Next, on the **Select Host** and **Select Data Source Type** windows, we leave the default values and click **Next**. On the **Select Data Source** window we click the **Browse button** on the right, we select the **Forensics02.dd** file, and we click **Next**. Finally, we click **Next** once again on the **Configure Ingest** window, and then **Finished**. Once the loading bar on the bottom right is fully loaded, the window should be looking like this.

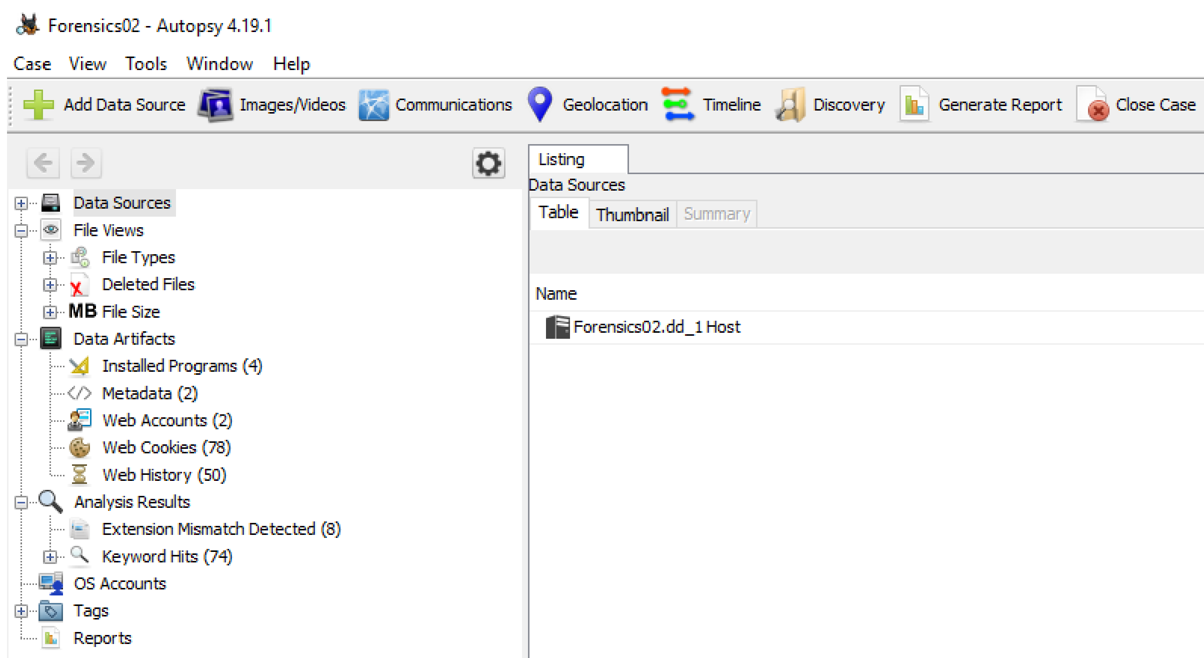


Figure 158 Forensics02 Home Screen

Enumeration of the **Web History** tab on the left side folding menu, reveals the website **http://custom-repository:8080/flag.zip**. It seems that the user has downloaded the file **flag.zip** from the website **http://custom-repository:8080**.

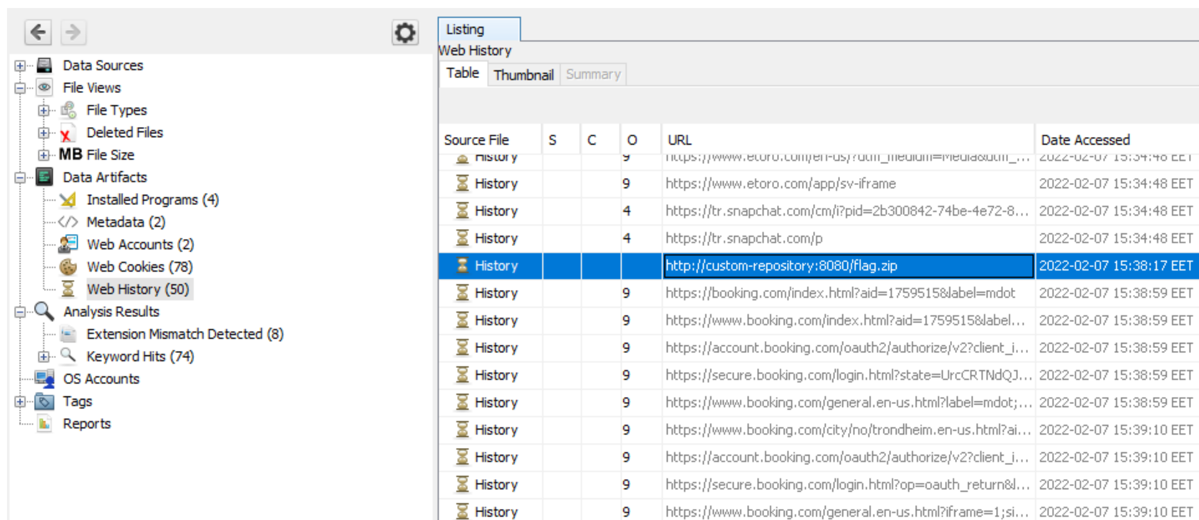


Figure 159 Forensics02 Web History

Searching common directories like **Downloads** and **Documents** won't reveal this file. However, enumeration of the **Deleted Files** reveals the file **flag.zip**. Under the **Data Sources -> File Views -> File Types -> By Extension -> Archives**, we can spot the file **f0212240_flag.zip**. We right click on it and select **Extract File(s)**.

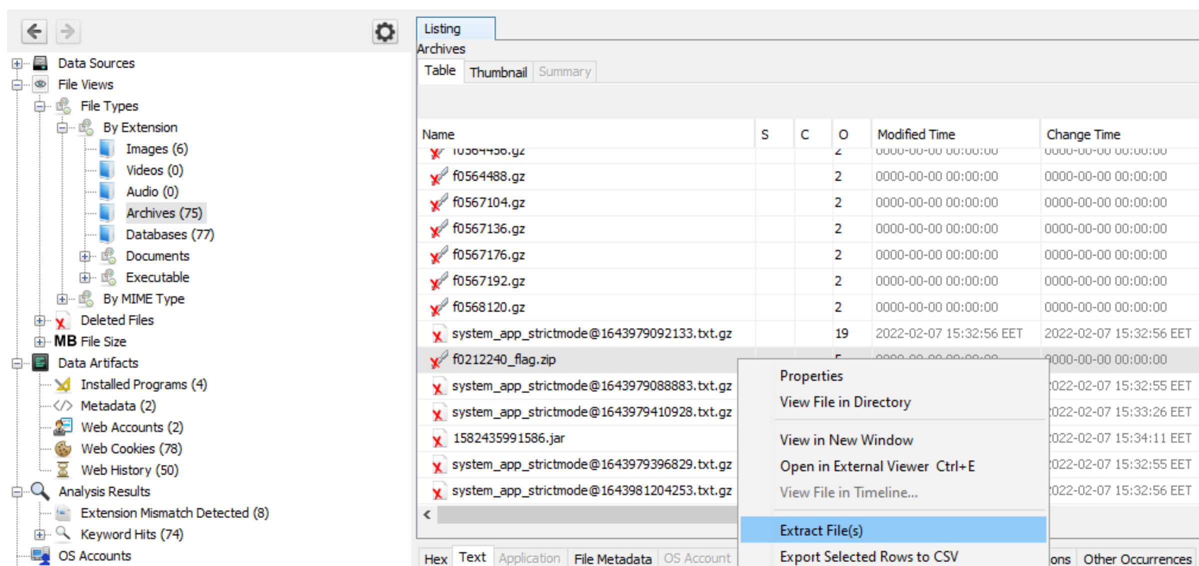


Figure 160 Forensics02 Extract Deleted File

We save it on the Desktop, and then we right click on it and select **Extract All**.

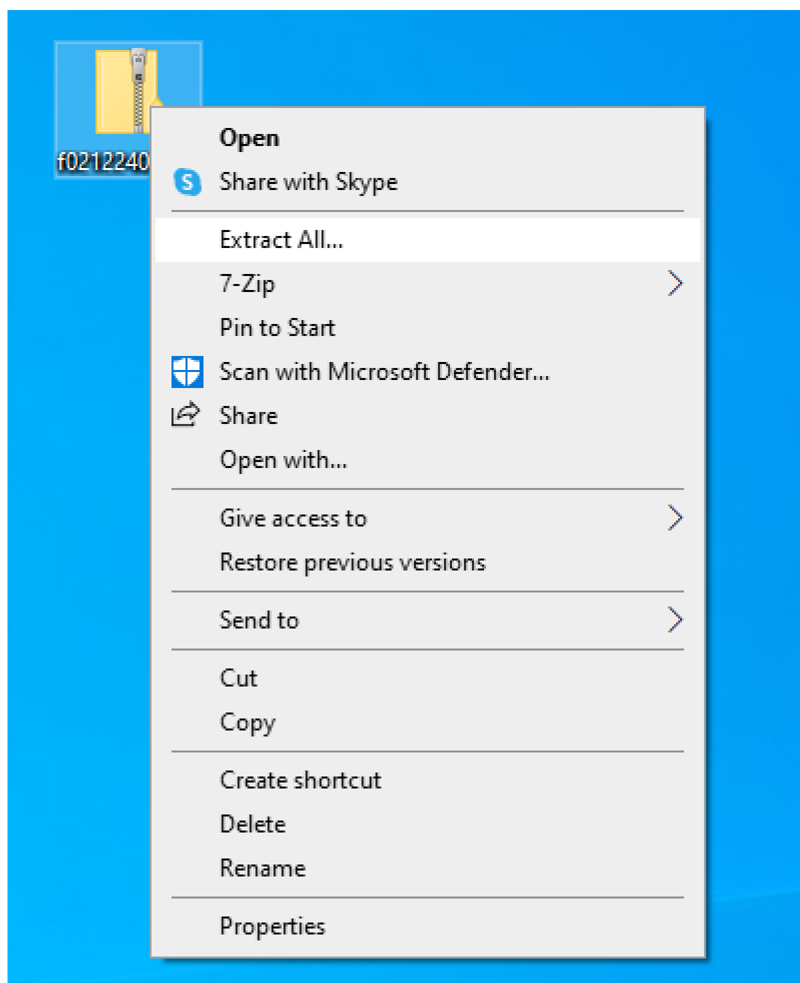


Figure 161 Forensics02 Decompress Extracted File

Once that's done, we double click on the extracted file in order to read the content.

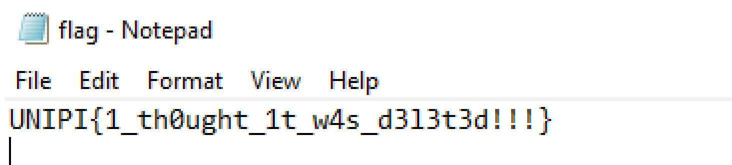


Figure 162 Forensics02 Flag

Flag: UNIPI{1_th0ught_1t_w4s_d313t3d!!!}

Conclusion

As we realise, smartphones these years are used massively to serve various of our daily purposes and thus, sensitive personal and corporate information is stored in these devices. This is the main reason that application security assessment teams must continuously train to stay up to date with the latest security trends, while at the same time individuals should have a good knowledge of what to pay attentions to, and use the application carefully.

In this thesis we discussed about the methodologies that one must follow in order to conduct application security assessment. We also analysed in depth some of the most common penetration testing techniques, while we got a good understanding of the purposes of each assessment technique.

It is now clear that in order to be able to conduct an application security assessment, first we have to understand some basic things about the Android OS and how it works, as well as to understand the application's structure.

Reading this thesis, we also learned how to set up and deploy a security training lab, based on the CTF approach, while we got an idea of how CTF events are contributing to the training in general, by providing a complete hands-on training experience in compare to the theoretical content that is provided by other online platforms.

By reading this thesis and finishing the Lab, one will be able to understand the need for cyber security training, the structure of an android phone and android apps, how to assess android applications, how to use tools to analyze the apps and automate the assessment procedures, how to do Android forensics, and how to create detailed writeups when completing a CTF security challenge.

In conclude, this project apart from raising the awareness of the mobile security, it also gives the learner a complete, methodical hands-on approach on how to start the Android security application assessment, while it can also keep up enterprise teams with the latest mobile security trend.

References

1. “Mobile Operating System Market Share Worldwide," [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide. [Accessed 2022].

2. “Check Point's Mobile Security Report 2021," [Online]. Available: https://blog.checkpoint.com/2021/04/12/check-points-mobile-security-report-2021-almost-every-organization-experienced-a-mobile-related-attack-in-2020/. [Accessed 2022].

3. “Application Fundamentals," [Online]. Available: https://developer.android.com/guide/components/fundamentals. [Accessed 2022].

4. “Dalvik (software)," [Online]. Available: https://en.wikipedia.org/wiki/Dalvik_(software). [Accessed 2022].

5. “Application Fundamentals," [Online]. Available: https://developer.android.com/guide/components/fundamentals. [Accessed 2022].

6. “.APK File Extension," [Online]. Available: https://fileinfo.com/extension/apk. [Accessed 2022].

7. “.DEX File Extension," [Online]. Available: https://fileinfo.com/extension/dex. [Accessed 2022].

8. “Capture the flag (cybersecurity)," [Online]. Available: https://en.wikipedia.org/wiki/Capture_the_flag_(cybersecurity). [Accessed 2022].

9. “CTF Categories," [Online]. Available: https://emaragkos.gr/ctfs/ctf-categories/. [Accessed 2022].

10. “Important Directories," [Online]. Available: https://www.hackthebox.com/blog/intro-to-mobile-pentesting#important_directories. [Accessed 2022].

11. “.APK File Extension," [Online]. Available: https://fileinfo.com/extension/apk. [Accessed 2022].

12. "App Manifest Overview," [Online]. Available: https://developer.android.com/guide/topics/manifest/manifest-intro. [Accessed 2022].

13. “Reverse-Engineering," [Online]. Available: https://searchsoftwarequality.techtarget.com/definition/reverse-engineering. [Accessed 2022].

14. “What is Static Analysis?," [Online]. Available: https://www.securecodewarrior.com/blog/what-is-static-analysis. [Accessed 2022].

15. “Getting started with Frida on Android Apps," [Online]. Available: https://payatu.com/blog/amit/Getting%20_started_with_Frida. [Accessed 2022].

16. “Extracting the APK from the Device," [Online]. Available: https://www.hackthebox.com/blog/intro-to-mobile-pentesting#extracting_the_apk_from_the_device. [Accessed 2022].

17. “OWASP Mobile Top 10," [Online]. Available: https://apprize.best/security/pentesting/6.html. [Accessed 2022].

18. “What is Autopsy," [Online]. Available: https://www.hackthebox.com/blog/intro-to-mobile-pentesting#forensics. [Accessed 2022].

19. “Rooting an Android Device," [Online]. Available: “BusyBox," [Online]. Available: https://busybox.net. [Accessed 2022].. [Accessed 2022].

20. “BusyBox," [Online]. Available: https://busybox.net. [Accessed 2022].

21. “dd (Unix) Utility," [Online]. Available: https://en.wikipedia.org/wiki/Dd_(Unix). [Accessed 2022

22. "15 best Android emulators for PC and Mac of 2022," [Online]. Available: https://www.androidauthority.com/best-android-emulators-for-pc-655308/. [Accessed 2022].

23. “Corellium," [Online]. Available: https://www.corellium.com. [Accessed 2022].

24. “Create and manage virtual devices," [Online]. Available: https://developer.android.com/studio/run/managing-avds. [Accessed 2022].

25. “Android Debug Bridge (adb)," [Online]. Available: https://developer.android.com/studio/command-line/adb. [Accessed 2022].

26. “Most Complete ADB Cheat Sheet," [Online]. Available: https://www.automatetheplanet.com/adb-cheat-sheet/. [Accessed 2022].

27. "Parrot OS," [Online]. Available: https://parrotsec.org. [Accessed 2022].

28. "Why CTFd?," [Online]. Available: https://ctfd.io/about/. [Accessed 2022].

29. "Docker (software)," [Online]. Available: https://en.wikipedia.org/wiki/Docker_(software). [Accessed 2022].

30. "Docker Container," [Online]. Available: https://www.docker.com/resources/what-container. [Accessed 2022].

31. “Overview of Docker Compose," [Online]. Available: https://docs.docker.com/compose/. [Accessed 2022].

32. “Docker image," [Online]. Available: https://searchitoperations.techtarget.com/definition/Docker-image. [Accessed 2022].

33. “Naumachia," [Online]. Available: https://github.com/nategraf/Naumachia. [Accessed 2022].

34. "Apktool," [Online]. Available: https://ibotpeaches.github.io/Apktool/. [Accessed 2022].

35. "Write Permissions," [Online]. Available: https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05h-testing-platform-interaction. [Accessed 2022].

36. "Shared Preferences," [Online]. Available: https://developer.android.com/training/data-storage/shared-preferences. [Accessed 2022].

37. "Encrypted Shared Preferences," [Online]. Available: https://developer.android.com/reference/androidx/security/crypto/EncryptedSharedPreferences. [Accessed 2022].

38. "What Is an APK File," [Online]. Available: https://www.makeuseof.com/tag/what-is-apk-file/. [Accessed 2022].

39. "Dalvik Executable format," [Online]. Available: https://source.android.com/devices/tech/dalvik/dex-format.html. [Accessed 2022].

40. "JAR (file format)," [Online]. Available: https://en.wikipedia.org/wiki/JAR_(file_format). [Accessed 2022].

41. "JADX," [Online]. Available: https://github.com/skylot/jadx. [Accessed 2022].

42. "Shrink, obfuscate, and optimize your app," [Online]. Available: https://developer.android.com/studio/build/shrink-code. [Accessed 2022].

43. ".SMALI File Extension," [Online]. Available: https://fileinfo.com/extension/smali. [Accessed 2022].

44. “App security best practices," [Online]. Available: https://developer.android.com/topic/security/best-practices. [Accessed 2022].

45. “Get started with the NDK," [Online]. Available: https://developer.android.com/ndk/guides. [Accessed 2022].

46. “.SO File Extension," [Online]. Available: https://fileinfo.com/extension/so. [Accessed 2022].

47. “Ghidra Software Reverse Engineering Framework," [Online]. Available: https://github.com/NationalSecurityAgency/ghidra. [Accessed 2022].

48. “Frida, Dynamic instrumentation toolkit," [Online]. Available: https://frida.re. [Accessed 2022].

49. “JNI Frida Hook," [Online]. Available: https://github.com/Areizen/JNI-Frida-Hook. [Accessed 2022].

50. "Burp Suite," [Online]. Available: https://www.pluralsight.com/paths/web-security-testing-with-burp-suite. [Accessed 2022].

51. "Android Network Security Configuration," [Online]. Available: https://developer.android.com/training/articles/security-config. [Accessed 2022].

52. "Autopsy," [Online]. Available: https://www.autopsy.com. [Accessed 2022].